NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

POPULATING THE SOFTWARE DATABASE

by

Tuan Anh Nguyen

March 1996

Thesis Co-Advisors: Luqi, V. Berzins

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)

2. REPORT DATE

March 1996

3. REPORT TYPE AND DATES COVERED

Master's Thesis

4. TITLE AND SUBTITLE

POPULATING THE SOFTWARE DATABASE

5. FUNDING NUMBERS

6. AUTHOR

Nguyen, Tuan A.

7. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSOR/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense of the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The cost of software development could be reduced if relevant reusable software components could be retrieved efficiently. The few libraries currently in existence have no standard method for selecting components germane to the intended application. This thesis focuses on the actual formation and population of library components for an improved software library model proposed in [Ref. 1]. This library would provides the codes for users to implement the desired system in CAPS environment.

The work reported here consists of: identifying candidate reusable components from the Booch Ada Library - by manually inspecting over 500 components; converting the components into a CAPS-compatible format based on the Prototyping System Description Language (PSDL) via Ada-PSDL converter program; creating algebraic specifications to match the semantic description of each component manually; and manually organizing the library into a data structure based on the multi-level filtering concept.

This work provides (1): the base and guidelines for the (a) criteria for a reusable component; (b) process of inspecting and importing components into CAPS reusable component library; (2): 75 reusable components to be released with CAPS 95 and used to test the user interface for retrieval via multi-level filtering. The process of populating reusable components is time intensive due to various manual processes. Inspecting and converting each component sometimes takes up to an hour for each. Current tools available can be rewritten, i.e. the PSDL-Ada converter, to fully automate this process in accordance with the base and guidelines.

14. SUBJECT TERMS
CAPS, Software Reuse, Software Base, Semantic Checking, Syntactic Checking

15. NUMBER OF PAGES 316

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	**UNCLASSIFIED**	**UNCLASSIFIED**	**UL**

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std 239-18

Approved for public release; distribution is unlimited

POPULATING THE SOFTWARE DATABASE

Tuan Anh Nguyen
Lieutenant, United States Naval Reserve
B.S.E.E., State University of New York at Buffalo, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1996

Author: ______________________
Tuan A. Nguyen

Approved by: ______________________
Luqi, Thesis Advisor

____
V. Berzins, Co-Advisor

____
T. Lewis, Chairman, Department of Computer Science

ABSTRACT

The cost of software development could be reduced if relevant reusable software components could be retrieved efficiently. The few libraries currently in existence have no standard method for selecting components germane to the intended application. This thesis focuses on the actual formation and population of library components for an improved software library model proposed in [Ref. 1]. This library would provides the codes for users to implement the desired system in CAPS environment.

The work reported here consists of: identifying candidate reusable components from the Booch Ada Library - by manually inspecting over 500 components; converting the components into a CAPS-compatible format based on the Prototyping System Description Language (PSDL) via Ada-PSDL converter program; creating algebraic specifications to match the semantic description of each component manually; and manually organizing the library into a data structure based on the multi-level filtering concept.

This work provides (1): the base and guidelines for the (a) criteria for a reusable component; (b) process of inspecting and importing components into CAPS reusable component library; (2): 75 reusable components to be released with CAPS 95 and used to test the user interface for retrieval via multi-level filtering. The process of populating reusable components is time intensive due to various manual processes. Inspecting and converting each component sometimes takes up to an hour for each. Current tools available can be rewritten, i.e. the PSDL-Ada converter, to fully automate this process in accordance with the base and guidelines.

TABLE OF CONTENTS

I. INTRODUCTION

The need for code reuse has not been addressed adequately in both the academic and business world. In the business world, most organizations appear to offer primitive incentives to encourage a culture of reuse. However, very few organizations explicitly encourage programmers to reuse code, or to write code that is reused. Reuse is preached more often than it is practiced. In the academic world, the word has been used but the teaching and the practices are also limited.

One of the reasons for this is the lack of methods for effectively finding the components needed for each application and lack of component libraries organized to support such methods. With the current trend of software development, prototyping tools seem to be the key for rapid developing applications, going from design to actual implementation with executable code. This idea of reusable code is instrumental to this prototyping concept. The Department of Defense has long endorsed a programming language that is rigid in structure, for safety of operation and most important of all the reusability of codes. Ada is the standard language of the DOD culture. The purpose of this thesis is to provide a library of reusable Ada components for the Computer Aided Prototyping System (CAPS), an ongoing research project at the Naval Postgraduate School.

A. WHY REUSE?

Each year, billions of dollars are spent on computer software. Much of this effort is spent on creating and testing new source code. In order to save money, increase productivity, and improve reliability, the Department of Defense is constructing

repositories of reusable software components that can be used across applications. A great percentage of a typical program is composed of potentially reusable code [Ref. 1] and [Ref. 2]. It is desirable to make use of existing code whenever possible. This action can significantly reduce the amount of time to develop the software. With prototyping software such as CAPS, reusable code can enhance the process of rapid application development.

This approach can be summarized as follows:

- Cost savings.
- Early payback.
- Manpower savings.
- Technology leverage and risk mitigation.
- Reliability.

B. COMPUTER AIDED PROTOTYPING SYSTEM

The Computer Aided Prototyping System is a software engineering tool for developing prototype models of hard real-time embedded systems [Ref. 3] and [Ref. 6]. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototype System Description Language (PSDL), which provides facilities for modeling timing and control constraints within a software system [Ref. 4]. It is a development environment, implemented in the form of an integrated collection of tools, linked together by a user-interface as shown in Figure 1 [Ref. 5].

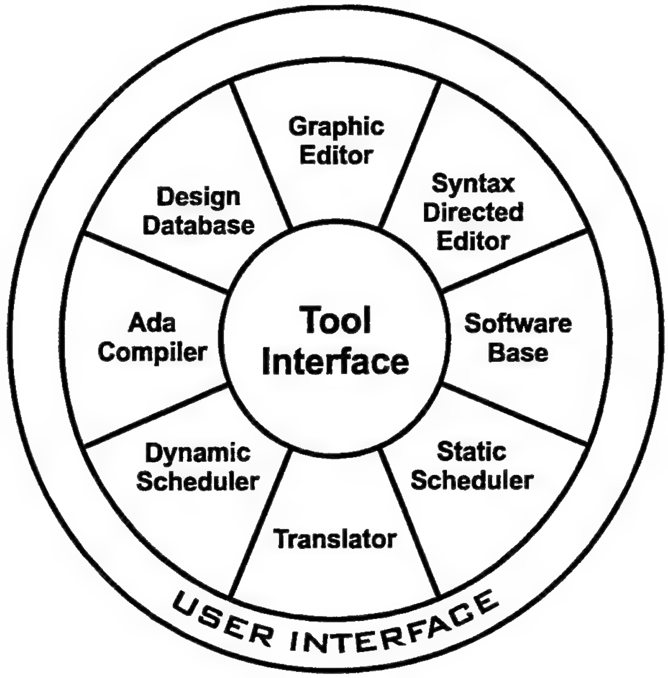

Figure 1. CAPS Functionality Overview Diagram

The library collected in this thesis is part of the Software Base component of the CAPS functionality.

C. ORGANIZATION OF CHAPTERS

Chapter II reviews the basic concepts and terms relevant to the current research of CAPS and its implementation. Chapter III focuses on the implementation of the database component of CAPS and the data structure and retrieval method for these reusable components. Chapter IV concludes the research and discusses the user interface of the software base component of CAPS.

II. BACKGROUND AND PREVIOUS RESEARCH

This chapter describes some technical background about CAPS to include PSDL. Characteristics of reusable components and methods of retrieval are the two primary topics of this section. Various previous research and current systems are also discussed.

A. CAPS DESIGN AND COMPONENTS

CAPS is an integrated environment aimed at rapid prototyping hard real-time embedded systems [Ref. 5] and [Ref. 6]. CAPS tools include an Ada Compiler, Design Database, Graphic Editor, Syntax Directed Editor, Software Base, Static Scheduler, Dynamic Scheduler, and Translator as shown in Figure 1. Each of these components provides specific functions in the development of the software.

B. PSDL

PSDL is a text and graphics based language designed to express the specifications of real-time systems. It is based on a graphic model of vertices and edges, in which the vertices represent operators, or software processes, and the edges represent the conceptual flow of data from one operator to another. Each vertex and edge may have associated timing constraint, and the vertices may have associated control constraints.

Formally, the model used is that of an augmented graph, $G = (V,E, T(v),C(v))$ where G is the graph, V is the set of vertices, E is the set of edges, $T(v)$ represents the timing constraints for the vertices, and $C(v)$ represents the control constraints for the vertices.

Conceptually, PSDL operators may contain other operators to support the principle of abstraction. Effectively, the prototype may be expressed as a flat graph, or a

one level graph containing all the atomic operators and their streams. An atomic operator is one that is implemented in a programming language, vice a composite operator consisting of other operators and streams.

For example, the following diagram shows a PSDL prototype:

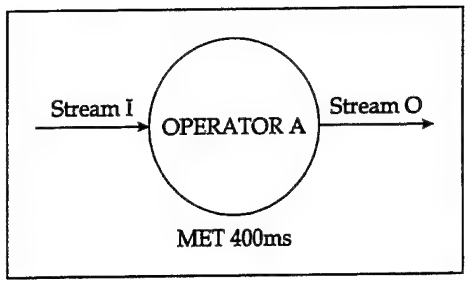

Figure 2. Example of PSDL Graph

Figure 2 represents an operation modeled by the Operator A that accepts one item from Stream I, it performs some operation on the data, and outputs Stream O. The Maximum Execution Time (MET), this is the maximum possible time the operator may take to execute the task, defined as 400 milliseconds.

Operator A can further be decomposed as shown in Figure 3 below:

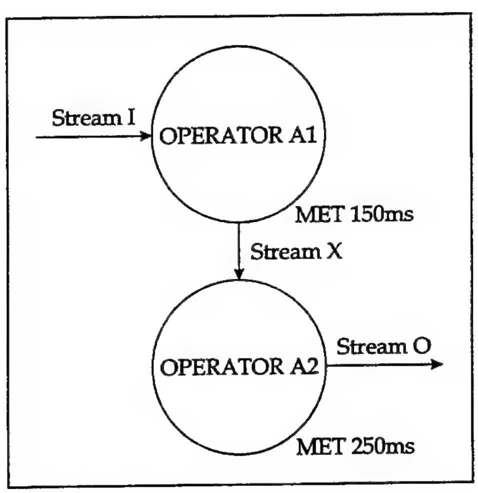

Figure 3. Decomposition of Operator A

Operator A is a composite operator, while Operator A1 and Operator A2 are atomic operators, implemented in Ada or some other language. The timing and control constraints on these atomic operators must be consistent with those of their parent operator. In a single processor the combined METs of these atomic operators cannot be greater than their parent. Operator A is really not needed for implementation of this prototype; it serves as an abstraction of the functionality of the children operators. More information about PSDL can be founded in [Ref. 8] and [Ref. 9].

C. OBJ3 AND ALGEBRAIC SPECIFICATION

OBJ3 is implemented in Common Lisp, and is based on ideas from order sorted equational logic and parameterised programming. OBJ3 provides mixfix syntax (prefix, suffix, and infix), flexible subsorts (subtypes in Ada language), parameterised modules, views, and most important term rewriting modulo associativity, commutativity, and identity. OBJ was originally designed in 1976 by Dr. Goguen [Ref. 10].

In OBJ3, an algebraic specification of objects consists of two parts: a signature and a set of axioms. The signature defines the sorts (or types) being specified, the operation symbols, and the axioms define their functionality in an object. The signature is denoted as (S, Σ) where S and Σ are a sort set and an operation symbol set, respectively. The axioms are expressed as equations describing the semantics of an object.

D. HASSE DIAGRAM

A Hasse diagram is a graphical representation of a partial ordering relation, for which the following properties hold:

- reflexive
- anti-symmetric
- transitive

For example: the Hasse diagram for ($\{1,2,3,4\},\leq$) is shown in Figure 4 below.

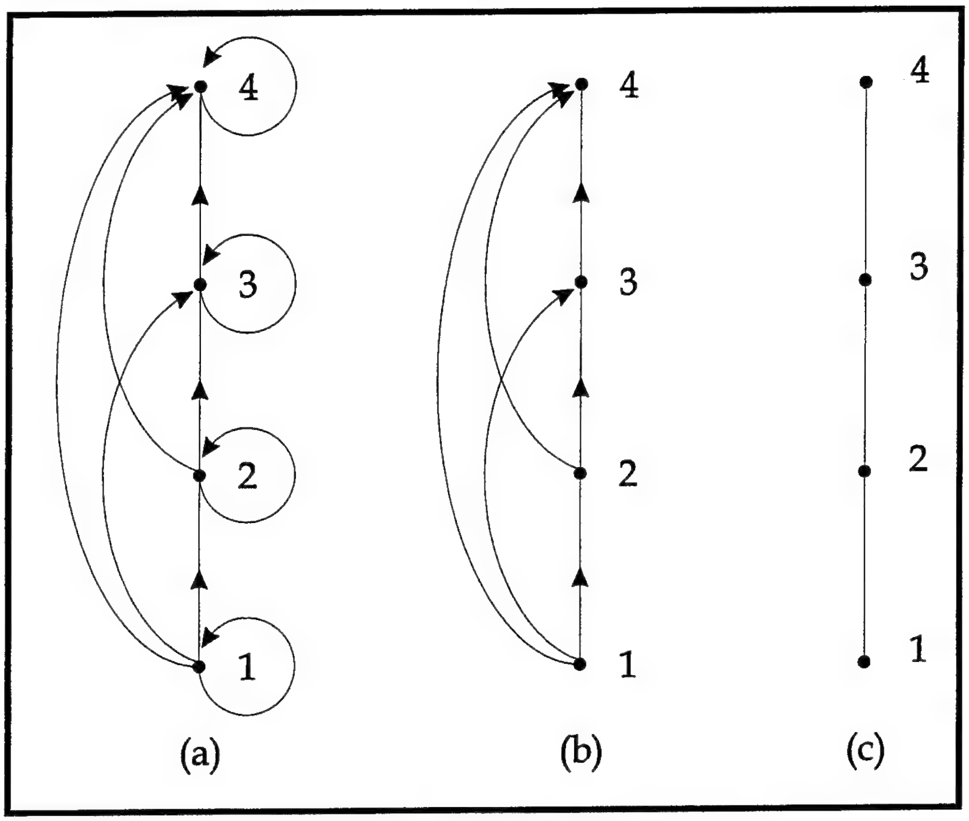

Figure 4. Constructing the Hasse Diagram for ($\{1,2,3,4\},\leq$)

This relation is called partial ordering. In Figure 4(a), the arrows indicated the relation among the members, since all members hold the reflexive property, the circle loops can be eliminated as shown in Figure 4(b), furthermore, since it is a partial ordering, all arrows implied by transitivity can be removed, as shown in Figure 4(c). This

concept can be extended to partition the software base, in which profile codes define partitions that are represented as a Hasse diagram.

E. PROFILE MATCHING

The computation for parameter matching would be very expensive if it was necessary to try all possible combinations of functions and data types with those components. For example, if a query has a function f: AAB -> B and a component has a function g: BA -> A, these two functions cannot be possibly be matched, thus there is no need to compute this combination. The purpose of profile matching is to speed up parameter matching. Profile matching is actually an efficient approximation of signature matching. A profile is a sequence of numbers that describes how data types are associated with an operation. It is defined as follows [Ref. 1]:

- The first integer is the total number of occurrences of sorts (data types).
- If the total number of sort groups, $N > 0$, then the second to $(1 + N)^{th}$ integers are the cardinalities of the sort groups, in descending order.
- The $(2 + N)^{th}$ integer is the cardinality of the unrelated sort group.
- The $(3+ N)^{th}$ integer is:

  0 if the value sort is different from any of the argument sorts; and

  1 if the value sort belongs to some sort group.

Sort groups are bags consisting of two or more sort occurrences from the rank of the operation that are related under the relation $\equiv$, which is the transitive-symmetric closure of the ordering $\leq$ on sorts.

Unrelated sort group is a set of all sort occurrences that are not in any sort group.

For example:

Operation	Profile Code
-> A	110
AB -> C	330
AA -> B	3210
ABBCA -> C	622201
CCAAB->B	622201

Table 1. Example of Profile Code

F. CHARACTERISTICS OF A REUSABLE COMPONENT

A reusable software component should exhibit the best characteristics of any good piece of software. Specifically, it should be:

- maintainable
- efficient
- reliable
- understandable

and of course, correct. However, there are some important characteristics specific to reuse. They should have the following major characteristics:

- generality
- definiteness
- transferability
- retrievability
- sufficiency
- completeness
- primitiveness

Generality and **Definiteness**: for example, a component supplying elementary real functions such as max, min, floor, and ceiling is a good candidate for reusability, because these operators are well understood and are applicable to a wide range of problems; this address the issue of definiteness. However, to facilitate its reuse, we must take care to construct such a component independent of the peculiarities of any application, for example, the representation of floating-point numbers. Ideally, we should factor out such dependencies and achieve generality. The Ada language has a mechanism to implement this characteristic, namely, generics and instantiation.

Transferability and **Retrievability**: primarily dealing at the level of source code, not object code. Writing a component as an Ada generic package facilitates transferability, for here we have a mechanism that can capture many of the relevant parts of an abstraction. However, the management of a library with a large number of components can be a great concern. The larger the number of components the higher the cost of finding a matching component.

Sufficiency: the component captures enough characteristics of the abstraction to permit meaningful interaction with the object.

Completeness: the component interface captures all characteristics of the component. Whereas sufficiency implies a minimal collection of meaningful operations, a complete set of operations is one that covers all aspects of the underlying abstraction. For example, the abstraction of a set includes the notion of cardinality. It is not necessary to include an operation that returns the cardinality of a set; we can interact with a set without this capability. However, we should include this operation to enhance the completeness of the abstraction. Completeness is a subjective measure and in fact can be

overdone. Supplying all meaningful operations for a particular abstraction is not only overwhelming for the user, but generally unnecessary, since many high-level operations can be composed from low-level ones. For this reason, It is suggested that component operations be primitive.

Primitiveness: operations that can be implemented only with access to the underlying representation of the object. Thus, adding an item to a set is primitive, because there is no other way to implement this operation unless the underlying representation is visible. However, adding four items to a set is not primitive since it can be implemented with the adding one item iteratively [Ref. 11].

G. SOFTWARE LIBRARIES

1. Asset Source for Software Engineering Technology (ASSET)

ASSET is a software reuse library and reuse information exchange available to software developers in government, industry, and education. ASSET is sponsored by ARPA's STARS (Software Technology for Adaptable, Reliable Systems) Program to serve as a national resource for the advancement of software reuse across the DoD. The ASSET library, located in Morgantown, WV, is connected to the Internet allowing world-wide access to reusable software assets. ASSET'S goals are to create a focal point for software reuse information exchange, to advance the technology of software reuse processes and to provide an electronic marketplace for reusable software products, and stimulate a national software reuse industry.

2. Reusable Ada Package for Information System Development (RAPID)

The RAPID project is an ongoing effort in the DoD. The objective of RAPID is to provide software engineers with quick access to reusable Ada packages in the

information system domain. The system performs reusable component classification, storage and retrieval.

3. Common Ada Missile Package (CAMP)

The CAMP project is also sponsored by the DoD to create a software engineering system of reusable software library of components. The system is directed toward software for missile systems and uses Ada language for its reusable components.

4. Operation Support System (OSS)

The OSS is an ongoing project aimed at developing and integrated software engineering environment. The system is being developed at the Naval Ocean System Center. One of the goals of the project is to establish a Naval software library of reusable software components.

H. METHODS OF RETRIEVAL

1. Keyword Search Method

This is the most crude method, however simplest of all. There is no data structure in storing these components. The user, in essence, is using a primitive grep UNIX command to search for a word that associated with a component. The useful components found by this method is extremely poor when the number of components in a library is large since the set of retrieved components is relatively large. This requires the user to browse through all the found components and decide which of the components is appropriate for usage. There is no way of placing the syntactic and semantic information in this method. However, from informal survey of current programmers in the private industry, this method is very popular. This may not be a surprise due to the fact that there is no standard in retrieving reusable components.

2. Artificial Intelligence Methods

Artificial Intelligence methods include [Ref. 9] and [Ref. 10], and some recent work by Henninger [Ref. 14], which uses a knowledge base and statistical information to retrieve reusable components, based on keyword search from texts describing the components. However, because the characterization of the component behavior is completely informal, the behavior is unpredictable [Ref. 15].

3. Multi-Level Filtering Method

This method is proposed in [Ref. 1], in which a combination of retrieval processes are used. The process is represented as follows:

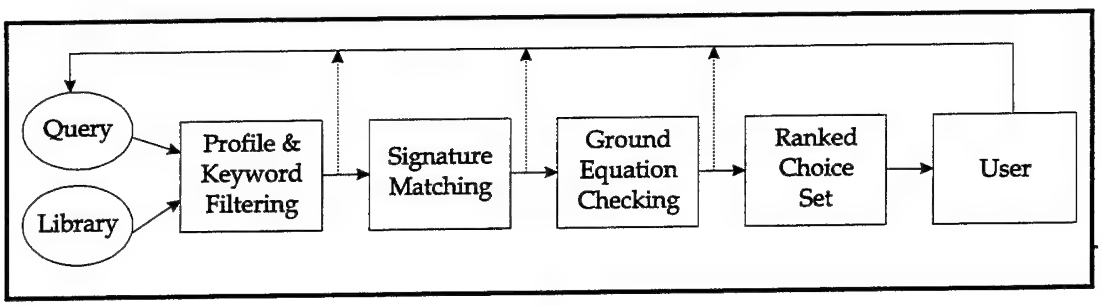

Figure 5. Model for the Multi-Level Filtering Process

In this method, search is organized as a series of increasingly stringent filters on candidate components. We first filter components by comparing their signatures with that of the query. This is accomplished by signature matching, which looks for maps that translate the type and function symbols of the query into corresponding type and function symbols of candidate components. A first stage of signature filtering can compare pre-computed syntactic profiles of components with the profile of the query. These profiles are special data structures that support an efficient approximation of signature matching. The key property of a profile is that two operation signatures cannot have a syntactic

match unless their computed profiles are equal. Signature matches can be partial, in that only part of the functionality the user seeks may actually be available. The profile of an abstract data type is a bag containing the profile codes of its operations. In a partial signature match, a subset of the query profile is contained in the stored component's profile. Traditional search methods, such as keyword search, could also be used as early filters. Profile matching should be followed by full signature matching.

Semantic filters rank components by how well they satisfy the equations in the query. In this process, equations that are logical consequences of the query specification are translated through the signature matches into equations whose proof is attempted in the candidate specifications. This whole process can be made iterative.

III. DESIGN AND CONCEPTS

A. BOOCH LIBRARY

The Booch library divided into three categories: data structure, tools, and subsystems. A data structure is a component that denotes an object or class of objects characterized as an abstract state machine or an abstract data type. A tool is a component that denotes an algorithmic abstraction targeted to an object or class of objects. A subsystem is a component that denotes a logical collection of cooperating structures and tools. Each category is further divided into subcategories as shown in Figure 6 below.

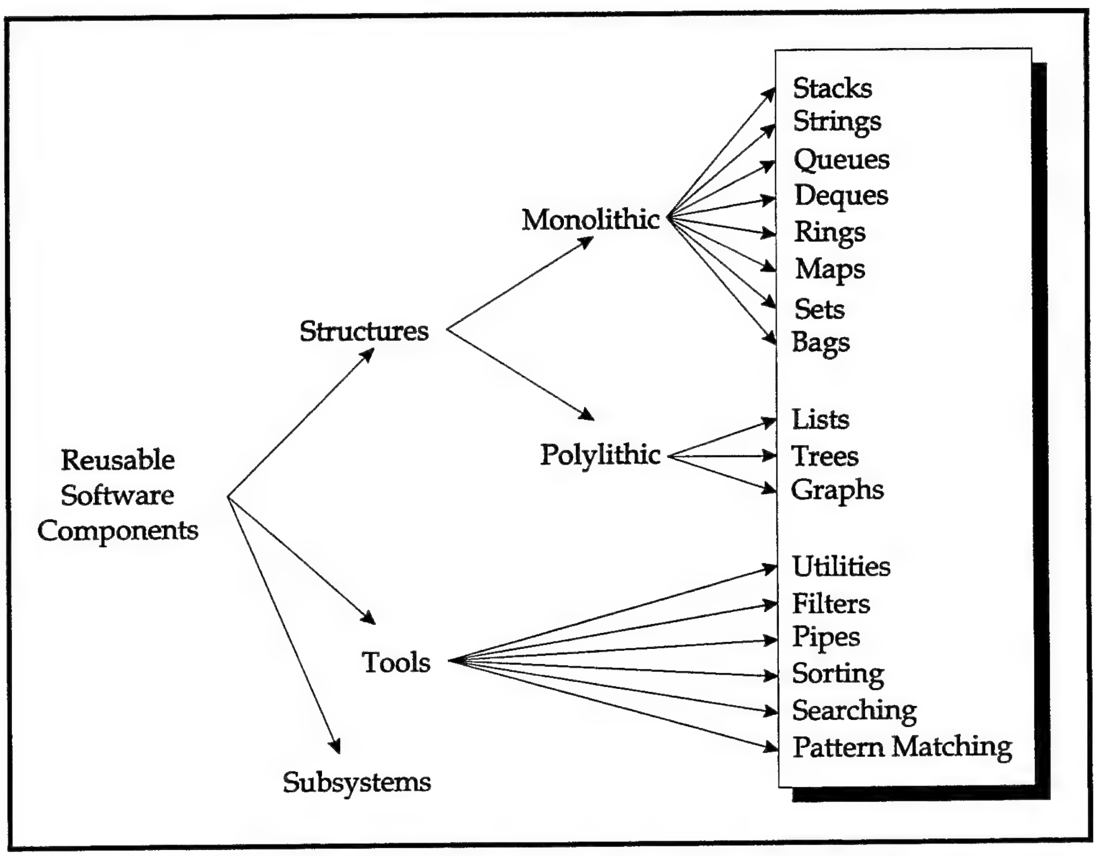

Figure 6. Booch Library

Monolithic the structure is always treated as a single unit and that individual parts of the structure can not be manipulated.

Polylithic the structure is composed of individual parts that can be manipulated.

There are over 500 components in the Booch library in many different forms. It often happens that there is a software part that we want to reuse, but it is not exactly in the right form [Ref. 16]. Figure 7 below presents the forms of reusable software component that have been found to be common across many applications [Ref. 11].

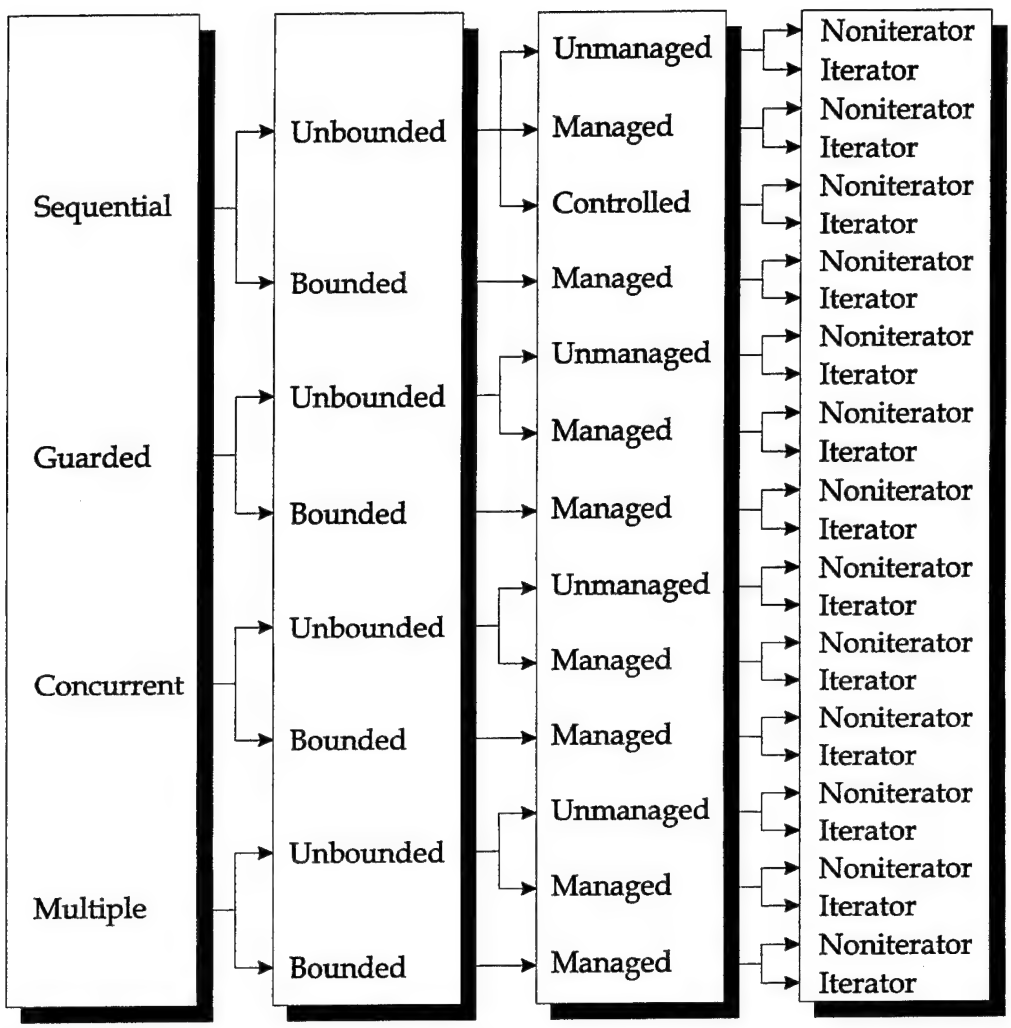

Figure 7. The forms of a reusable software component

Sequential The semantics of an object are preserved only in the presence of one thread of control for each instance of the type.

Guarded The semantics of an object are preserved in the presence of multiple threads of control, if mutual exclusion is enforced by all clients of the object.

Concurrent The semantics of an object are preserved in the presence of multiple threads of control, and mutual exclusion is enforced by the object itself. Access by multiple clients is sequentialized.

Multiple The semantics of an object are preserved in the presence of multiple threads of control, and mutual exclusion is enforced by the object itself. Multiple simultaneous readers are permitted, but writers are sequentialized.

Bounded Denotes that the size of the object is static.

Unbounded Denotes that the size of the object is dynamic.

Unmanaged Automatic garbage collection is the responsibility of the underlying run time system and compiler.

Managed Garbage collection is provided by the component itself, and the type is used only by a single task.

Controlled Garbage collection is provided by the sequential component itself even if the type is used by multiple tasks.[1]

Noniterator An iterator is not provided for this object.

---

[1] Sequential controlled means several tasks can each have a private instance of the type.

Iterator An iterator is provided for this object.

Together, these forms offer a total of 26 meaningful combinations. The Appendix lists the imported components.

The components in the library conform to the following file name convention: assuming the file name of the component is stackssbmn.

Description	**File Name**
Ada specifications	vstackssbmn.**a**
Ada implementation	**b**stackssbmn.**a**
PSDL	vstackssbmn.**psdl**
OBJ3 specifications	vstackssbmn.**obj**
Profile code	vstackssbmn.**code**

Table 2. Example of file name convention

There are 75 components imported into this library. These components are the samples of each of the data structure components in the Booch library. This should give a broad base number of the components for the reusable components.

B. POPULATING PROCESS

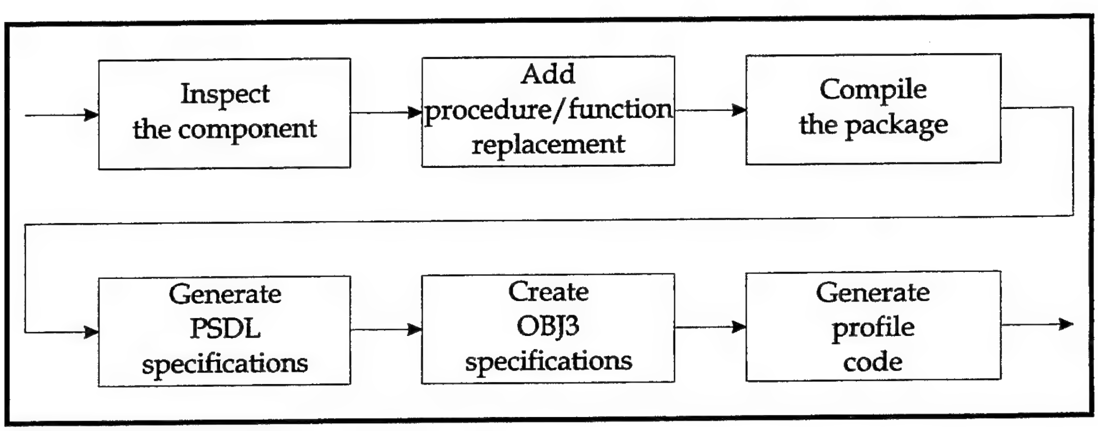

Figure 8. Populating Process

Components must be manually inspected for reusability criteria listed in Chapter II. In the CAPS system, a PSDL specification is an integrated part of a reusable component. By adding procedure versions of functions, PSDL specifications can be readily generated by a converter written by [Ref. 17].

Each step of the populating process, shown in Figure 8, is illustrated in this section by an example. An example of the first step, adding procedure/function replacement, follows:

SPECIFICATIONS

```
generic
    type Item is private;
package Stack_Sequential_Bounded_Managed_Iterator is

    type Stack(The_Size : Positive) is limited private;

    procedure Copy  (From_The_Stack : in     Stack;
                     To_The_Stack   : in out Stack);
    procedure Clear (The_Stack      : in out Stack);
    procedure Push  (The_Item       : in     Item;
                     On_The_Stack   : in out Stack);
    procedure Pop   (The_Stack      : in out Stack);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left       : in Stack;
                        Right      : in Stack;
                        Result     : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result     : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal  (Left      : in Stack;
                        Right     : in Stack) return
Boolean;

    function Depth_Of  (The_Stack : in Stack) return
Natural;
```

```
    function Is_Empty (The_Stack : in Stack) return Boolean;

    function Top_Of   (The_Stack : in Stack) return Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Stack : in Stack);

    Overflow   : exception;
    Underflow  : exception;

private
    type Items is array(Positive range <>) of Item;
    type Stack(The_Size : Positive) is
        record
            The_Top   : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
     end Stack_Sequential_Bounded_Managed_Iterator;
```

IMPLEMENTATION

```
package body Stack_Sequential_Bounded_Managed_Iterator is

    procedure Copy (From_The_Stack : in     Stack;
                    To_The_Stack   : in out Stack) is
    begin
        if From_The_Stack.The_Top > To_The_Stack.The_Size
then
            raise Overflow;
        else
            To_The_Stack.The_Items(1 ..
From_The_Stack.The_Top) :=
                From_The_Stack.The_Items(1 ..
From_The_Stack.The_Top);
            To_The_Stack.The_Top := From_The_Stack.The_Top;
        end if;
    end Copy;

    procedure Clear (The_Stack : in out Stack) is
    begin
        The_Stack.The_Top := 0;
    end Clear;

    procedure Push (The_Item     : in     Item;
                    On_The_Stack : in out Stack) is
    begin
        On_The_Stack.The_Items(On_The_Stack.The_Top + 1) :=
The_Item;
        On_The_Stack.The_Top := On_The_Stack.The_Top + 1;
    exception
        when Constraint_Error =>
```

```
            raise Overflow;
    end Push;

    procedure Pop (The_Stack : in out Stack) is
    begin
        The_Stack.The_Top := The_Stack.The_Top - 1;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing procedures with functions

    procedure Is_Equal (Left   : in Stack;
                        Right  : in Stack;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Depth_Of (The_Stack : in Stack;
                        Result    : out Natural) is
    begin
        Result := Depth_Of(The_Stack);
    end Depth_Of;

    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean) is
    begin
        Result := Is_Empty(The_Stack);
    end Is_Empty;

    procedure Top_Of (The_Stack : in Stack;
                      Result    : out Item) is
    begin
        Result := Top_Of(The_Stack);
    end Top_Of;

--  end of modification

    function Is_Equal (Left  : in Stack;
                       Right : in Stack) return Boolean is
    begin
        if Left.The_Top /= Right.The_Top then
            return False;
        else
            for Index in 1 .. Left.The_Top loop
                if Left.The_Items(Index) /=
Right.The_Items(Index) then
                    return False;
                end if;
            end loop;
            return True;
```

```
        end if;
    end Is_Equal;

    function Depth_Of (The_Stack : in Stack) return Natural
is
    begin
        return The_Stack.The_Top;
    end Depth_Of;

    function Is_Empty (The_Stack : in Stack) return Boolean
is
    begin
        return (The_Stack.The_Top = 0);
    end Is_Empty;

    function Top_Of (The_Stack : in Stack) return Item is
    begin
        return The_Stack.The_Items(The_Stack.The_Top);
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    procedure Iterate (Over_The_Stack : in Stack) is
        Continue : Boolean;
    begin
        for The_Iterator in reverse 1 ..
Over_The_Stack.The_Top loop
            Process(Over_The_Stack.The_Items(The_Iterator),
Continue);
            exit when not Continue;
        end loop;
    end Iterate;

end Stack_Sequential_Bounded_Managed_Iterator;
```

This procedure is necessary to match the code interface conventions of the current implemementation of CAPS. The next step is to generate the PSDL specification for the component. The converter program will generate the PSDL automatically with the following command:

ada2psdl filename (without any extension)

The output file will have the same name as the file name with the psdl extension.

The generated file for the above example follows:

PSDL

```
TYPE Stack_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To_The_Stack : Stack
    EXCEPTIONS
      Overflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
  END

  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Stack,
      Right : Stack
    OUTPUT
      Result : Boolean
```

```
  END

  OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Item
    EXCEPTIONS
      Underflow
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue
: out[t : Boolean]]
    INPUT
      Over_The_Stack : Stack
  END

END
IMPLEMENTATION ADA Stack_Sequential_Bounded_Managed_Iterator
END
```

Each procedure in Ada specifications is associated with an operator in PSDL. The input and output streams in PSDL correspond to the procedure input/output parameters.

The package must then be re-compiled for quality assurance.

OBJ3 specifications are created next in accordance with the guideline in Chapter II. The following is an example of this step (for the previous Ada specifications):

STACK OJB3 SPECIFICATION:

```
obj STACK[X :: TRIV] is sort Stack .
   protecting NAT .

*** constructors

   op create      :                 -> Stack .
   op copy        : Stack Stack -> Stack .
   op clear       :        Stack -> Stack .
   op push        : Elt    Stack -> Stack .
   op pop         :        Stack -> Stack .

*** accessors

   op isequal    : Stack Stack -> Bool .
   op depthof    :       Stack -> Nat .
   op isempty    :       Stack -> Bool .
   op topof      :       Stack -> Elt .

*** exceptions

   op underflow  : -> Stack .
   op underflow  : -> Elt .

*** variables declaration

   var S S1 : Stack .
   var E E1 : Elt .

*** axioms

   eq clear(S) = create .

   eq copy(S,S1) = S .

   eq pop(create) = underflow .
   eq pop(push(E,S)) = S .

   eq isequal(S,S1) = S == S1 .

   eq depthof(S) = if S == create then 0
                   else 1 + depthof(pop(S)) fi .

   eq isempty(S) = S == create .

   eq topof(create) = underflow .
   eq topof(push(E,S)) = E .

endo
```

The next step is to create the profile code:

From either the Ada specifications or PSDL:

```
procedure Copy   (From_The_Stack : in     Stack;
                  To_The_Stack   : in out Stack);
```

has the signature AB -> B

- ° first digit is the number of sort occurrences: 3
- ° the number of sort groups is 1 thus N = 1;
- ° $(1 + N)^{th}$ digit is the cardinality of the sort group
- ° second digit (1 + 1) is : 2 since $|[B,B]| = 2$
- ° third digit (2 + 1) is : 1 since [A] is the only unrelated sort group
- ° fourth digit (3 + 1) is : 1 since B belongs to the sort group

thus: profile(Copy) = 3211

```
procedure Clear (The_Stack       : in out Stack);
```

Clear: A -> A has profile 2201

```
procedure Push   (The_Item       : in     Item;
                  On_The_Stack   : in out Stack);
```

Push: AB -> B has profile 3211

```
procedure Pop    (The_Stack      : in out Stack);
```

Pop: A -> A has profile 2201

```
procedure Is_Equal (Left       : in Stack;
                    Right      : in Stack;
                    Result     : out Boolean);
```

Is_Equal: AB -> C has profile 330

```
procedure Depth_Of (The_Stack  : in Stack;
                    Result     : out Natural);
```

Depth_Of: A -> B has profile 220

```
procedure Is_Empty (The_Stack : in Stack;
                    Result    : out Boolean);
```

Is_Empty: A -> B has profile 220

```
procedure Top_Of (The_Stack   : in Stack;
                  Result      : out Item);
```

Top_Of: A -> B has profile 220

Summary:

Operation	Signature	Profile Code
Copy	AB -> B	3211
Clear	A -> A	2201
Push	AB -> B	3211
Pop	A -> A	2201
Is_Equal	AB -> C	330
Is_Empty	A -> B	220
Depth_Of	A -> B	220
Top_Of	A -> B	220

Table 3. Summary of Stack Profile Code

The profile codes from these components will then be partitioned and represented by a Hasse diagram to optimize the multi-filtering retrieval method.

IV. CONCLUSIONS AND FUTURE RESEARCH

This chapter summarizes the concept and the process of populating the software base. Lessons learned and suggestions for future research are also mentioned in this section.

A. ACCOMPLISHMENT

This thesis has described the process of populating the software base and relevant method for retrieval, namely, multi-level filtering concept. The components selected comprise the base library listed in the Appendix, which can be used for future study and testing of the multi-level filtering process. This process is labor intensive and many automation issues should be investigated further. Preliminary study of the retrieval has been very promising [Ref. 18].

B. LESSONS LEARNED

The process is time intensive. Not all components can be reused. The primary difference between engineering reusable components, i.e. nuts and bolts, and software engineering is continuity in dimension. A nut will be manufactured only in certain dimensions such as 5/8" but a graphical representation of a nut in software engineering can be any size.

The writing of the OBJ3 specifications associated with each component is the most difficult task of all. OBJ3 is a functional language, however Ada components are written with procedures. Thus multiple out parameters cannot be directly implemented. The rationale for using OBJ3 is to attach the semantics of the operations to each data type. By attaching this specification to a component the system can refine the retrieving

process. The user can accurately retrieve the matched component via this specification. However, the user, most of the time, does not search for an exact component, just for an approximation of the component. The user must and should inspect and modify the component found to meet his/her requirements. Thus completely detailed OBJ3 specifications may not be that critical. For example: a bounded stack will have an overflow exception in its specification. This aspect cannot be easily handled during semantic matching. Consequently, the user must supply the size parameter during instantiation. This exception can be omitted in the OBJ3 specification because the semantic matching process cannot use the information. A more appropriate treatment of the exception is to include an informal explanation sufficient to guide the user in instantiating the size bound. The informal description part of the PSDL specification can be used for this purpose.

C. FUTURE RESEARCH

1. Graphical User Interface

A graphical user interface can make the retrieval process less error prone. The user would not need to be an expert in how the software base works. This interface will increase productivity.

2. CAPS and the Internet

Currently, CAPS can be used on a local area network Unix environment or a stand alone Unix workstation. There is a plan to implement CAPS on another microprocessor base, namely, the Intel architecture microprocessor. However, CAPS can be used across platforms via the Internet. JavaScript, based on the Java language (a derivative of the C++ language), and the Internet can make this possible. JavaScript extends the

programmatic capabilities of a typical Internet browser, i.e. Netscape, to a wide range of authors and is easy enough for anyone who can compose Hyper Text Markup Language (HTML). JavaScript can be used to glue HTML, inline plug-ins, and Java applets (applications) to each other. It provides the ability to change images, play different sounds, and more in response to specified events such as a user mouse click or screen exit and entry.

The JavaScript language resembles Java, but without Java's static typing and strong type checking. JavaScript supports most of Java's expression syntax and basic control flow constructs. In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a run-time system based on a small number of primitive types. The members of numeric, boolean, and string types can be expressed literally.

Primitive types can be composed into objects by setting properties with the assignment operator. JavaScript also supports functions, again without any declarative requirements beyond the need to distinguish a function definition from other sentences in the language. Functions can be properties of objects, executing as loosely-typed methods.

JavaScript complements Java by exposing useful properties of Java applets to script authors. JavaScript scripts embedded in HTML documents can get and set exposed properties in order to query the state or alter the performance of an applet or plug-in.

Java is an extension language designed, in particular, for fast execution and type safety. (Type safety is reflected by being unable to cast a Java int into an object reference or to get at private memory by corrupting Java bytecodes). Java's strong typing also increases compilation efficiency of Java bytecode to machine code.

Java programs consist exclusively of classes and their methods. Java's requirements for declaring classes, writing methods, and ensuring type safety make programming more complex than JavaScript authoring. Java's inheritance and strong typing also tend to require tightly coupled object hierarchies.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically-typed languages like HyperTalk and Dbase. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

In summary, JavaScript can be used to implement World Wide Web access to various aspects of CAPS. For example, a graphical user interface, written in JavaScript, can enable the user to retrieve a component from the Software Base library. JavaScript can provide dialog boxes, error messages, and help systems. These features enable the user to interact with CAPS via the Internet without having to fully implement CAPS locally. Multimedia (video and audio) can be distributed over the Internet as a marketing tool for CAPS.

LIST OF REFERENCES

1. Joseph Goguen, Doan Nguyen, José Meseguer, Luqi, Du Zhang, Valdis Berzins. "Software Component Search", *Journal of Systems Integration*, Vol. 6, No. 1, 1996, pp. 93-134.

2. Cox, B. J., "Planning the Software Industrial Revolution", *IEEE Software*, September, 1990, pp. 22-30.

3. Luqi, V. Berzins, "Rapidly Prototyping Real-Time Sytems", *IEEE Software*, September 1988, pp. 25-36.

4. Luqi, "Real-Time Constraints in a Rapid Prototyping Language", *Journal of Computer Languages*, Vol. 18, No. 2, Spring 1993, pp. 77-103.

5. Jim Brockett, "The Computer Aided Prototyping System (CAPS)", *A CAPS Tutorial*, 1994.

6. Luqi and Ketabchi, M., "A Computer Aided Prototyping System", *IEEE Software*, March 1988, pp. 66-72.

7. Luqi, "Computer Aided Software Prototyping", *IEEE Computer*, September 1991, pp. 111-112.

8. Luqi, Berzins, V., Yeh, R. T., "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, October 1988, Vol. 14, No. 10, pp. 1409-1423.

9. Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer*, May 1989, pp. 13-25.

10. Joseph Goguen. "Abstract Errors for Abstract Data Types", *Proceedings, First IFIP Working Conference on Formal Description of Programming Concepts*, MIT press, 1977, pp. 21.1-21.32.

11. Grady Booch, "Software Components with Ada", The Benjamin/Cummings Publishing Company, Inc., 1987.

12. G. Fischer, Scott Henninger, and D. Redmiles. "Cognitive Tools for Locating and Comprehending Software Components", *Proceedings, 13th International Conference on Software Engineering*, May 1991.

13. Horowitz, E., and J. Munson. "An Expansive View of Reusable Software", *IEEE Transactions on Software Engineering*, Vol. SE-10(5), p. 479.

14. E. Ostertag, J. Hendler, Rubin Prieto-Diaz, and C. Braun. Computing Similarity in a Reusable Library System. *ACM Transaction on Software Engineering and Methodology*, July 1992, pp. 205-228.

15. Scott Henninger and Cornelia Boldyreff. "Software Engineer's Reference Book". Butterworth-Heinemann, 1991.

16. Goguen, J. "Parameterized Programming", *IEEE Transactions on Software Engineering*, Vol. SE-10(5), September 1984, pp. 474.

17. Christopher S. Eagle, "Tools for Storage and Retrieval of Ada Software Components in a Software Base", Master Thesis, March 1995.

18. Doan Nguyen, "Architectural Model for Software Component Search", Ph. D. Dissertation, December 1995.

APPENDIX - LIBRARY COMPONENTS

BOOCH LIBRARY COMPONENTS

The following lists are grouped by major component class.

Bags

1	Bag_Simple_Sequential_Bounded_Managed_Iterator
2	Bag_Simple_Sequential_Bounded_Managed_Noniterator
3	Bag_Simple_Sequential_Unbounded_Managed_Iterator
4	Bag_Simple_Sequential_Unbounded_Managed_Noniterator
5	Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator
6	Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator

Lists

1	List_Double_Bounded_Managed
2	List_Double_Unbounded_Managed
3	List_Double_Unbounded_Unmanaged
4	List_Single_Bounded_Managed
5	List_Single_Unbounded_Managed
6	List_Single_Unbounded_Unmanaged

Maps

1	Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator
2	Map_Simple_Noncached_Sequential_Bounded_Managed_Noniterator
3	Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
4	Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator
5	Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator

Queues

1	Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator
2	Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator
3	Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator
4	Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
5	Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator
6	Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator
7	Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator
8	Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
9	Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
10	Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
11	Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator
12	Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
13	Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator
14	Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
15	Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
16	Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
17	Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
18	Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator

Rings

1	Ring_Sequential_Bounded_Managed_Iterator
2	Ring_Sequential_Bounded_Managed_Noniterator
3	Ring_Sequential_Unbounded_Managed_Iterator
4	Ring_Sequential_Unbounded_Managed_Noniterator
5	Ring_Sequential_Unbounded_Managed_Iterator
6	Ring_Sequential_Unbounded_Managed_Noniterator

Sets

1	Set_Simple_Sequential_Bounded_Managed_Iterator
2	Set_Simple_Sequential_Bounded_Managed_Noniterator
3	Set_Simple_Sequential_Unbounded_Managed_Iterator
4	Set_Simple_Sequential_Unbounded_Managed_Noniterator
5	Set_Simple_Sequential_Unbounded_Unmanaged_Iterator
6	Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator

Sorts & Searchs

1	Binary_Search
2	Binary_Insertion_Search
3	Buble_Sort
4	Heap_Sort
5	Natural_Merge_Sort
6	Ordered_Sequential_Search
7	Poly_Sort
8	Quick_Sort
9	Radix_Sort
10	Sequential_Search
11	Shaker_Sort
12	Shell_Sort
13	Straight_Insertion_Sort
14	Straight_Selection_Sort

Stacks

1	Stack_Sequential_Bounded_Managed_Iterator
2	Stack_Sequential_Unbounded_Managed_Noniterator
3	Stack_Sequential_Unbounded_Managed_Iterator
4	Stack_Sequential_Unbounded_Unmanaged_Noniterator
5	Stack_Sequential_Unbounded_Unmanaged_Iterator

Storage

1	Storage_Sequence

Strings

1	String_Sequential_Unbounded_Controlled_Iterator
2	String_Sequential_Unbounded_Managed_Iterator
3	String_Sequential_Bounded_Unmanaged_Noniterator
4	String_Sequential_Unbounded_Unmanaged_Noniterator

Trees

1	Tree_Arbitrary_Double_Bounded_Unmanaged
2	Tree_Arbitrary_Double_Unbounded_Unmanaged
3	Tree_Arbitrary_Single _Bounded_Unmanaged
4	Tree_Arbitrary_Single _Unbounded_Unmanaged

BAG OBJ3 SPECIFICATIONS

```
obj BAG[X :: TRIV] is sort Bag .
   protecting NAT .

*** constructors

   op create           :               -> Bag .
   op copy             :       Bag Bag -> Bag .
   op clear            :           Bag -> Bag .
   op add              :       Elt Bag -> Bag .
   op remove           :       Elt Bag -> Bag .
   op union            : Bag Bag Bag -> Bag .
   op intersection     : Bag Bag Bag -> Bag .
   op difference       : Bag Bag Bag -> Bag .

*** accessors

   op isequal          : Bag Bag -> Bool .
   op extentof         :     Bag -> Nat .
   op uniqueextentof   :     Bag -> Nat .
   op isempty          :     Bag -> Bool .
   op isamember        : Elt Bag -> Bool .
   op isasubset        : Bag Bag -> Bool .
   op isapropersubset  : Bag Bag -> Bool .

*** exceptions

   op overflow         :         -> Bag .
   op itemisnotinbag   :         -> Bag .

*** variables declaration

   var B B1 B2 : Bag .
   var E E1    : Elt .

*** axioms

   eq copy(B,B1) = B .

   eq clear(B) = create .

   eq remove(E,create) = itemisnotinbag .
   eq remove(E,add(E1,B1)) = if E == E1 then B1 else
add(E1,remove(E,B1)) fi .

   eq union(B,create,B1) = B .
   eq union(B,add(E1,B1),B2) = add(E1,union(B,B1,B2)) .

   eq intersection(B,create,B1) = create .
   eq intersection(B,add(E1,B1),B2) = if isamember(E1,B) then
add(E1,intersection(B,B1,B2)) else intersection(B,B1,B2) fi .

   eq difference(B,create,B1) = B .
   eq difference(create,B,B1) = B .
   eq difference(B,add(E1,B1),B2) = if isamember(E1,B) then
difference(remove(E1,B),B1,B2) else add(E1,difference(B,B1,B2)) fi .

   eq extentof(create) = 0 .
   eq extentof(add(E,B)) = 1 + extentof(B) .

   eq uniqueextentof(create) = 0 .
   eq uniqueextentof(add(E,B)) = if isamember(E,B) then
uniqueextentof(B) else 1 + uniqueextentof(B) fi .

   eq isempty(B) = B == create .

   eq isamember(E,create) = false .
   eq isamember(E,add(E1,B1)) = E == E1 or isamember(E,B1) .

   eq isasubset(create,B) = true .
   eq isasubset(add(E,B),B1) = if isamember(E,B1) then isasubset(B,B1)
else  false fi .

   eq isapropersubset(B,B1) = isasubset(B,B1) and extentof(B1) >
extentof(B) .

endo
```

BAGS PROFILE CODES

OPERATORS	*SIGNATURES*	*PROFILE CODES*
COPY	A B -> B	3211
CLEAR	A -> A	2201
ADD	A B -> B	3211
REMOVE	A B -> B	3211
UNION	A B C -> C	4231
INTERSECTION	A B C -> C	4231
DIFFERENCE	A B C -> C	4231
IS_EQUAL	A B -> C	330
EXTENT_OF	A -> B	220
UNIQUE_EXTENT_OF	A -> B	220
IS_EMPTY	A -> B	220
IS_A_MEMBER	A B -> C	330
IS_A_SUBSET	A B -> C	330
IS_A_PROPER_SUBSET	A B -> C	330

SET OF PROFILE: {4231,3211,2201,330,220}

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
obj BAG[X :: TRIV] is sort Bag .
   protecting NAT .

*** constructors

   op create           :             -> Bag .
   op copy             :     Bag Bag -> Bag .
   op clear            :         Bag -> Bag .
   op add              :     Elt Bag -> Bag .
   op remove           :     Elt Bag -> Bag .
   op union            : Bag Bag Bag -> Bag .
   op intersection     : Bag Bag Bag -> Bag .
   op difference       : Bag Bag Bag -> Bag .

*** accessors

   op isequal          : Bag Bag -> Bool .
   op extentof         :     Bag -> Nat .
   op uniqueextentof   :     Bag -> Nat .
   op isempty          :     Bag -> Bool .
   op isamember        : Elt Bag -> Bool .
   op isasubset        : Bag Bag -> Bool .
   op isapropersubset  : Bag Bag -> Bool .

*** exceptions

   op overflow         :         -> Bag .
   op itemisnotinbag   :         -> Bag .

*** variables declaration

   var B B1 B2 : Bag .
   var E E1    : Elt .

*** axioms

   eq copy(B,B1) = B .

   eq clear(B) = create .

   eq remove(E,create) = itemisnotinbag .
   eq remove(E,add(E1,B1)) = if E == E1 then B1 else
add(E1,remove(E,B1)) fi .

   eq union(B,create,B1) = B .
   eq union(B,add(E1,B1),B2) = add(E1,union(B,B1,B2)) .

   eq intersection(B,create,B1) = create .
   eq intersection(B,add(E1,B1),B2) = if isamember(E1,B) then
add(E1,intersection(B,B1,B2)) else intersection(B,B1,B2) fi .

   eq difference(B,create,B1) = B .
   eq difference(create,B,B1) = B .
   eq difference(B,add(E1,B1),B2) = if isamember(E1,B) then
difference(remove(E1,B),B1,B2) else add(E1,difference(B,B1,B2)) fi .

   eq extentof(create) = 0 .
   eq extentof(add(E,B)) = 1 + extentof(B) .

   eq uniqueextentof(create) = 0 .
   eq uniqueextentof(add(E,B)) = if isamember(E,B) then
uniqueextentof(B) else 1 + uniqueextentof(B) fi .

   eq isempty(B) = B == create .

   eq isamember(E,create) = false .
   eq isamember(E,add(E1,B1)) = E == E1 or isamember(E,B1) .

   eq isasubset(create,B) = true .
   eq isasubset(add(E,B),B1) = if isamember(E,B1) then isasubset(B,B1)
else  false fi .

   eq isapropersubset(B,B1) = isasubset(B,B1) and extentof(B1) >
extentof(B) .

endo
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Bag_Simple_Sequential_Bounded_Managed_Iterator is

    procedure Copy (From_The_Bag : in     Bag;
                    To_The_Bag   : in out Bag) is
    begin
        if From_The_Bag.The_Back > To_The_Bag.The_Size then
            raise Overflow;
        else
            To_The_Bag.The_Items(1 .. From_The_Bag.The_Back) :=
              From_The_Bag.The_Items(1 .. From_The_Bag.The_Back);
            To_The_Bag.The_Back := From_The_Bag.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Bag : in out Bag) is
    begin
        The_Bag.The_Back := 0;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Bag : in out Bag) is
    begin
        for Index in 1 .. To_The_Bag.The_Back loop
            if The_Item = To_The_Bag.The_Items(Index).The_Item then
                To_The_Bag.The_Items(Index).The_Count :=
                  To_The_Bag.The_Items(Index).The_Count + 1;
                return;
            end if;
        end loop;
        To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Item :=
The_Item;
        To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Count := 1;
        To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Bag : in out Bag) is
    begin
        for Index in 1 .. From_The_Bag.The_Back loop
            if The_Item = From_The_Bag.The_Items(Index).The_Item then
                if From_The_Bag.The_Items(Index).The_Count > 1 then
                    From_The_Bag.The_Items(Index).The_Count :=
                      From_The_Bag.The_Items(Index).The_Count - 1;
                else
                    From_The_Bag.The_Items(Index ..
                                           (From_The_Bag.The_Back -
1)) :=
                      From_The_Bag.The_Items((Index + 1) ..
                                             From_The_Bag.The_Back);
                    From_The_Bag.The_Back := From_The_Bag.The_Back -
1;
                end if;
                return;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Remove;

    procedure Union (Of_The_Bag : in     Bag;
                     And_The_Bag: in     Bag;
                     To_The_Bag : in out Bag) is
        To_Index : Natural;
        To_Back  : Natural;
    begin
        To_The_Bag.The_Items(1 .. Of_The_Bag.The_Back) :=
          Of_The_Bag.The_Items(1 .. Of_The_Bag.The_Back);
        To_The_Bag.The_Back := Of_The_Bag.The_Back;
        To_Back := To_The_Bag.The_Back;
        for And_Index in 1 .. And_The_Bag.The_Back loop
            To_Index := To_Back;
            while To_Index > 0 loop
                if To_The_Bag.The_Items(To_Index).The_Item =
                   And_The_Bag.The_Items(And_Index).The_Item then
                    exit;
                else
                    To_Index := To_Index - 1;
                end if;
            end loop;
            if To_Index = 0 then
                To_The_Bag.The_Items(To_The_Bag.The_Back + 1) :=
                  And_The_Bag.The_Items(And_Index);
                To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
            else
                To_The_Bag.The_Items(To_Index).The_Count :=
                  To_The_Bag.The_Items(To_Index).The_Count +
                  And_The_Bag.The_Items(And_Index).The_Count;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Bag  : in     Bag;
                            And_The_Bag : in     Bag;
                            To_The_Bag  : in out Bag) is
        And_Index : Natural;
    begin
        To_The_Bag.The_Back := 0;
        for Of_Index in 1 .. Of_The_Bag.The_Back loop
            And_Index := And_The_Bag.The_Back;
            while And_Index > 0 loop
                if Of_The_Bag.The_Items(Of_Index).The_Item =
                   And_The_Bag.The_Items(And_Index).The_Item then
                    if Of_The_Bag.The_Items(Of_Index).The_Count <
                       And_The_Bag.The_Items(And_Index).The_Count then
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Item
                          := Of_The_Bag.The_Items(Of_Index).The_Item;
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Count
                          := Of_The_Bag.The_Items(Of_Index).The_Count;
                        To_The_Bag.The_Back := To_The_Bag.The_Back +
1;
                    else
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Item
                          := Of_The_Bag.The_Items(Of_Index).The_Item;
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Count
                          :=
And_The_Bag.The_Items(And_Index).The_Count;
                        To_The_Bag.The_Back := To_The_Bag.The_Back +
1;
                    end if;
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Bag  : in     Bag;
                          And_The_Bag : in     Bag;
                          To_The_Bag  : in out Bag) is
        And_Index : Natural;
    begin
        To_The_Bag.The_Back := 0;
        for Of_Index in 1 .. Of_The_Bag.The_Back loop
            And_Index := And_The_Bag.The_Back;
            while And_Index > 0 loop
                if Of_The_Bag.The_Items(Of_Index).The_Item =
                   And_The_Bag.The_Items(And_Index).The_Item then
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
            if And_Index = 0 then
                To_The_Bag.The_Items(To_The_Bag.The_Back + 1) :=
                  Of_The_Bag.The_Items(Of_Index);
                To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
            elsif Of_The_Bag.The_Items(Of_Index).The_Count >
                  And_The_Bag.The_Items(And_Index).The_Count then
                To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Item
:=
                  Of_The_Bag.The_Items(Of_Index).The_Item;
                To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Count :=
                  Of_The_Bag.The_Items(Of_Index).The_Count -
                  And_The_Bag.The_Items(And_Index).The_Count;
                To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Difference;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left    : in Bag;
                        Right   : in Bag;
```

```
                    Result : out Boolean) is
begin
    Result := Is_Equal(Left,Right);
end Is_Equal;

procedure Extent_Of (The_Bag : in Bag;
                     Result  : out Natural) is
begin
    Result := Extent_Of(The_Bag);
end Extent_Of;

procedure Unique_Extent_Of (The_Bag : in Bag;
                            Result  : out Natural) is
begin
    Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;

procedure Number_Of (The_Item   : in Item;
                     In_The_Bag : in Bag;
                     Result     : out Positive) is
begin
        Result := Number_Of(The_Item,In_The_Bag);
end Number_Of;

procedure Is_Empty (The_Bag : in Bag;
                    Result  : out Boolean) is
begin
    Result := Is_Empty(The_Bag);
end Is_Empty;

procedure Is_A_Member (The_Item   : in Item;
                       Of_The_Bag : in Bag;
                       Result     : out Boolean) is
begin
    Result := Is_A_Member(The_Item,Of_The_Bag);
end Is_A_Member;

procedure Is_A_Subset (Left   : in Bag;
                       Right  : in Bag;
                       Result : out Boolean) is
begin
    Result := Is_A_Subset(Left,Right);
end Is_A_Subset;

procedure Is_A_Proper_Subset (Left   : in Bag;
                              Right  : in Bag;
                              Result : out Boolean) is
begin
    Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;

-- end of modification

function Is_Equal (Left  : in Bag;
                   Right : in Bag) return Boolean is
    Right_Index : Natural;
begin
    if Left.The_Back /= Right.The_Back then
        return False;
    else
        for Left_Index in 1 .. Left.The_Back loop
            Right_Index := Right.The_Back;
            while Right_Index > 0 loop
                if Left.The_Items(Left_Index).The_Item =
                   Right.The_Items(Right_Index).The_Item then
                    if Left.The_Items(Left_Index).The_Count /=
                       Right.The_Items(Right_Index).The_Count then
                        return False;
                    else
                        exit;
                    end if;
                else
                    Right_Index := Right_Index - 1;
                end if;
            end loop;
            if Right_Index = 0 then
                return False;
            end if;
        end loop;
        return True;
    end if;
end Is_Equal;

function Extent_Of (The_Bag : in Bag) return Natural is
    Count : Natural := 0;
begin
    for Index in 1 .. The_Bag.The_Back loop
        Count := Count + The_Bag.The_Items(Index).The_Count;
    end loop;
    return Count;
end Extent_Of;

function Unique_Extent_Of (The_Bag : in Bag) return Natural is
begin
    return The_Bag.The_Back;
end Unique_Extent_Of;

function Number_Of (The_Item   : in Item;
                    In_The_Bag : in Bag) return Positive is
begin
    for Index in 1 .. In_The_Bag.The_Back loop
        if The_Item = In_The_Bag.The_Items(Index).The_Item then
            return In_The_Bag.The_Items(Index).The_Count;
        end if;
    end loop;
    raise Item_Is_Not_In_Bag;
end Number_Of;

function Is_Empty (The_Bag : in Bag) return Boolean is
begin
    return (The_Bag.The_Back = 0);
end Is_Empty;

function Is_A_Member (The_Item   : in Item;
                      Of_The_Bag : in Bag) return Boolean is
begin
    for Index in 1 .. Of_The_Bag.The_Back loop
        if Of_The_Bag.The_Items(Index).The_Item = The_Item then
            return True;
        end if;
    end loop;
    return False;
end Is_A_Member;

function Is_A_Subset (Left  : in Bag;
                      Right : in Bag) return Boolean is
    Right_Index : Natural;
begin
    for Left_Index in 1 .. Left.The_Back loop
        Right_Index := Right.The_Back;
        while Right_Index > 0 loop
            if Left.The_Items(Left_Index).The_Item =
               Right.The_Items(Right_Index).The_Item then
                exit;
            else
                Right_Index := Right_Index - 1;
            end if;
        end loop;
        if Right_Index = 0 then
            return False;
        elsif Left.The_Items(Left_Index).The_Count >
              Right.The_Items(Right_Index).The_Count then
            return False;
        end if;
    end loop;
    return True;
end Is_A_Subset;

function Is_A_Proper_Subset (Left  : in Bag;
                             Right : in Bag) return Boolean is
    Total_Left_Count  : Natural := 0;
    Total_Right_Count : Natural := 0;
    Right_Index       : Natural;
begin
    for Left_Index in 1 .. Left.The_Back loop
        Right_Index := Right.The_Back;
        while Right_Index > 0 loop
            if Left.The_Items(Left_Index).The_Item =
               Right.The_Items(Right_Index).The_Item then
                exit;
            else
                Right_Index := Right_Index - 1;
            end if;
        end loop;
        if Right_Index = 0 then
            return False;
        elsif Left.The_Items(Left_Index).The_Count >
              Right.The_Items(Right_Index).The_Count then
            return False;
        end if;
        Total_Left_Count := Total_Left_Count +
                            Left.The_Items(Left_Index).The_Count;
    end loop;
    for Index in 1 .. Right.The_Back loop
        Total_Right_Count := Total_Right_Count +
                             Right.The_Items(Index).The_Count;
    end loop;
    if Left.The_Back < Right.The_Back then
        return True;
    elsif Left.The_Back > Right.The_Back then
        return False;
    else
        return (Total_Left_Count < Total_Right_Count);
    end if;
end Is_A_Proper_Subset;

procedure Iterate (Over_The_Bag : in Bag) is
    Continue : Boolean;
begin
    for The_Iterator in 1 .. Over_The_Bag.The_Back loop
        Process(Over_The_Bag.The_Items(The_Iterator).The_Item,
                Over_The_Bag.The_Items(The_Iterator).The_Count,
Continue);
        exit when not Continue;
    end loop;
end Iterate;

end Bag_Simple_Sequential_Bounded_Managed_Iterator;
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Bag : Bag
    OUTPUT
      From_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Unique_Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], The_Count : in[t :
Positive], Continue : out[t : Boolean]]
    INPUT
      Over_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

END

KEYWORDS: BAG
DESCRIPTIONS:  (Bag, Simple, Sequential, Bounded, Managed, Iterator)

IMPLEMENTATION ADA Bag_Simple_Sequential_Bounded_Managed_Iterator
END
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Bag_Simple_Sequential_Bounded_Managed_Noniterator is

    type Bag(The_Size : Positive) is limited private;

    procedure Copy         (From_The_Bag : in      Bag;
                            To_The_Bag   : in out Bag);
    procedure Clear        (The_Bag      : in out Bag);
    procedure Add          (The_Item     : in      Item;
                            To_The_Bag   : in out Bag);
    procedure Remove       (The_Item     : in      Item;
                            From_The_Bag : in out Bag);
    procedure Union        (Of_The_Bag   : in      Bag;
                            And_The_Bag  : in      Bag;
                            To_The_Bag   : in out Bag);
    procedure Intersection (Of_The_Bag   : in      Bag;
                            And_The_Bag  : in      Bag;
                            To_The_Bag   : in out Bag);
    procedure Difference   (Of_The_Bag   : in      Bag;
                            And_The_Bag  : in      Bag;
                            To_The_Bag   : in out Bag);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  7 April 1995
--  adding procedures to replace functions

    procedure Is_Equal             (Left         : in Bag;
                                    Right        : in Bag;
                                    Result       : out Boolean);
    procedure Extent_Of            (The_Bag      : in Bag;
                                    Result       : out Natural);
    procedure Unique_Extent_Of     (The_Bag      : in Bag;
                                    Result       : out Natural);
    procedure Is_Empty             (The_Bag      : in Bag;
                                    Result       : out Boolean);
    procedure Is_A_Member          (The_Item     : in Item;
                                    Of_The_Bag   : in Bag;
                                    Result       : out Boolean);

    procedure Is_A_Subset          (Left         : in Bag;
                                    Right        : in Bag;
                                    Result       : out Boolean);
    procedure Is_A_Proper_Subset (Left         : in Bag;
                                    Right        : in Bag;
                                    Result       : out Boolean);

--  end of modification

    function Is_Equal            (Left       : in Bag;
                                  Right      : in Bag)  return Boolean;
    function Extent_Of           (The_Bag    : in Bag)  return Natural;
    function Unique_Extent_Of    (The_Bag    : in Bag)  return Natural;
    function Number_Of           (The_Item   : in Item;
                                  In_The_Bag : in Bag)  return
Positive;
    function Is_Empty            (The_Bag    : in Bag)  return Boolean;
    function Is_A_Member         (The_Item   : in Item;
                                  Of_The_Bag : in Bag)  return Boolean;
    function Is_A_Subset         (Left       : in Bag;
                                  Right      : in Bag)  return Boolean;
    function Is_A_Proper_Subset (Left       : in Bag;
                                  Right      : in Bag)  return Boolean;

    Overflow          : exception;
    Item_Is_Not_In_Bag : exception;

private
    type Node is
        record
            The_Item  : Item;
            The_Count : Positive;
        end record;
    type Items is array(Positive range <>) of Node;
    type Bag(The_Size : Positive) is
        record
            The_Back  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Bag_Simple_Sequential_Bounded_Managed_Noniterator;
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Bag_Simple_Sequential_Bounded_Managed_Noniterator is

    procedure Copy (From_The_Bag : in     Bag;
                    To_The_Bag   : in out Bag) is
    begin
        if From_The_Bag.The_Back > To_The_Bag.The_Size then
            raise Overflow;
        else
            To_The_Bag.The_Items(1 .. From_The_Bag.The_Back) :=
              From_The_Bag.The_Items(1 .. From_The_Bag.The_Back);
            To_The_Bag.The_Back := From_The_Bag.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Bag : in out Bag) is
    begin
        The_Bag.The_Back := 0;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Bag : in out Bag) is
    begin
        for Index in 1 .. To_The_Bag.The_Back loop
            if The_Item = To_The_Bag.The_Items(Index).The_Item then
                To_The_Bag.The_Items(Index).The_Count :=
                  To_The_Bag.The_Items(Index).The_Count + 1;
                return;
            end if;
        end loop;
        To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Item :=
The_Item;
        To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Count := 1;
        To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Bag : in out Bag) is
    begin
        for Index in 1 .. From_The_Bag.The_Back loop
            if The_Item = From_The_Bag.The_Items(Index).The_Item then
                if From_The_Bag.The_Items(Index).The_Count > 1 then
                    From_The_Bag.The_Items(Index).The_Count :=
                      From_The_Bag.The_Items(Index).The_Count - 1;
                else
                    From_The_Bag.The_Items(Index ..
                                    (From_The_Bag.The_Back -
1)) :=
                      From_The_Bag.The_Items((Index + 1) ..
                                    From_The_Bag.The_Back);
                    From_The_Bag.The_Back := From_The_Bag.The_Back -
1;
                end if;
                return;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Remove;

    procedure Union (Of_The_Bag : in     Bag;
                     And_The_Bag: in     Bag;
                     To_The_Bag : in out Bag) is
        To_Index : Natural;
        To_Back  : Natural;
    begin
        To_The_Bag.The_Items(1 .. Of_The_Bag.The_Back) :=
          Of_The_Bag.The_Items(1 .. Of_The_Bag.The_Back);
        To_The_Bag.The_Back := Of_The_Bag.The_Back;
        To_Back := To_The_Bag.The_Back;
        for And_Index in 1 .. And_The_Bag.The_Back loop
            To_Index := To_Back;
            while To_Index > 0 loop
                if To_The_Bag.The_Items(To_Index).The_Item =
                   And_The_Bag.The_Items(And_Index).The_Item then
                    exit;
                else
                    To_Index := To_Index - 1;
                end if;
            end loop;
            if To_Index = 0 then
                To_The_Bag.The_Items(To_The_Bag.The_Back + 1) :=
                  And_The_Bag.The_Items(And_Index);
                To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
            else
                To_The_Bag.The_Items(To_Index).The_Count :=
                  To_The_Bag.The_Items(To_Index).The_Count +
                  And_The_Bag.The_Items(And_Index).The_Count;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Bag  : in     Bag;
                            And_The_Bag : in     Bag;
                            To_The_Bag  : in out Bag) is
        And_Index : Natural;
    begin
        To_The_Bag.The_Back := 0;
        for Of_Index in 1 .. Of_The_Bag.The_Back loop
            And_Index := And_The_Bag.The_Back;
            while And_Index > 0 loop
                if Of_The_Bag.The_Items(Of_Index).The_Item =
                   And_The_Bag.The_Items(And_Index).The_Item then
                    if Of_The_Bag.The_Items(Of_Index).The_Count <
                      And_The_Bag.The_Items(And_Index).The_Count then
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Item
                          := Of_The_Bag.The_Items(Of_Index).The_Item;
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Count
                          := Of_The_Bag.The_Items(Of_Index).The_Count;
                        To_The_Bag.The_Back := To_The_Bag.The_Back +
1;
                    else
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Item
                          := Of_The_Bag.The_Items(Of_Index).The_Item;
                        To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Count
                          :=
And_The_Bag.The_Items(And_Index).The_Count;
                        To_The_Bag.The_Back := To_The_Bag.The_Back +
1;
                    end if;
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Bag  : in     Bag;
                          And_The_Bag : in     Bag;
                          To_The_Bag  : in out Bag) is
        And_Index : Natural;
    begin
        To_The_Bag.The_Back := 0;
        for Of_Index in 1 .. Of_The_Bag.The_Back loop
            And_Index := And_The_Bag.The_Back;
            while And_Index > 0 loop
                if Of_The_Bag.The_Items(Of_Index).The_Item =
                   And_The_Bag.The_Items(And_Index).The_Item then
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
            if And_Index = 0 then
                To_The_Bag.The_Items(To_The_Bag.The_Back + 1) :=
                    Of_The_Bag.The_Items(Of_Index);
                To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
            elsif Of_The_Bag.The_Items(Of_Index).The_Count >
                  And_The_Bag.The_Items(And_Index).The_Count then
                To_The_Bag.The_Items(To_The_Bag.The_Back + 1).The_Item
:=
                  Of_The_Bag.The_Items(Of_Index).The_Item;
                To_The_Bag.The_Items(To_The_Bag.The_Back +
1).The_Count :=
                  Of_The_Bag.The_Items(Of_Index).The_Count -
                  And_The_Bag.The_Items(And_Index).The_Count;
                To_The_Bag.The_Back := To_The_Bag.The_Back + 1;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Difference;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left    : in Bag;
                        Right   : in Bag;
```

```
                          Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Bag : in Bag;
                         Result  : out Natural) is
    begin
        Result := Extent_Of(The_Bag);
    end Extent_Of;

    procedure Unique_Extent_Of (The_Bag : in Bag;
                                Result  : out Natural) is
    begin
        Result := Unique_Extent_Of (The_Bag);
    end Unique_Extent_Of;

    procedure Number_Of (The_Item   : in Item;
                         In_The_Bag : in Bag;
                         Result     : out Positive) is
    begin
            Result := Number_Of(The_Item,In_The_Bag);
    end Number_Of;

    procedure Is_Empty (The_Bag : in Bag;
                        Result  : out Boolean) is
    begin
        Result := Is_Empty(The_Bag);
    end Is_Empty;

    procedure Is_A_Member (The_Item   : in Item;
                           Of_The_Bag : in Bag;
                           Result     : out Boolean) is
    begin
        Result := Is_A_Member(The_Item,Of_The_Bag);
    end Is_A_Member;

    procedure Is_A_Subset (Left   : in Bag;
                           Right  : in Bag;
                           Result : out Boolean) is
    begin
        Result := Is_A_Subset(Left,Right);
    end Is_A_Subset;

    procedure Is_A_Proper_Subset (Left   : in Bag;
                                  Right  : in Bag;
                                  Result : out Boolean) is
    begin
        Result := Is_A_Proper_Subset(Left,Right);
    end Is_A_Proper_Subset;

--  end of modification

    function Is_Equal (Left  : in Bag;
                       Right : in Bag) return Boolean is
        Right_Index : Natural;
    begin
        if Left.The_Back /= Right.The_Back then
            return False;
        else
            for Left_Index in 1 .. Left.The_Back loop
                Right_Index := Right.The_Back;
                while Right_Index > 0 loop
                    if Left.The_Items(Left_Index).The_Item =
                       Right.The_Items(Right_Index).The_Item then
                        if Left.The_Items(Left_Index).The_Count /=
                           Right.The_Items(Right_Index).The_Count then
                            return False;
                        else
                            exit;
                        end if;
                    else
                        Right_Index := Right_Index - 1;
                    end if;
                end loop;
                if Right_Index = 0 then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Extent_Of (The_Bag : in Bag) return Natural is
        Count : Natural := 0;
    begin
        for Index in 1 .. The_Bag.The_Back loop
            Count := Count + The_Bag.The_Items(Index).The_Count;
        end loop;
        return Count;
    end Extent_Of;

    function Unique_Extent_Of (The_Bag : in Bag) return Natural is
    begin
        return The_Bag.The_Back;
    end Unique_Extent_Of;

    function Number_Of (The_Item   : in Item;
                        In_The_Bag : in Bag) return Positive is
    begin
        for Index in 1 .. In_The_Bag.The_Back loop
            if The_Item = In_The_Bag.The_Items(Index).The_Item then
                return In_The_Bag.The_Items(Index).The_Count;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Number_Of;

    function Is_Empty (The_Bag : in Bag) return Boolean is
    begin
        return (The_Bag.The_Back = 0);
    end Is_Empty;

    function Is_A_Member (The_Item   : in Item;
                          Of_The_Bag : in Bag) return Boolean is
    begin
        for Index in 1 .. Of_The_Bag.The_Back loop
            if Of_The_Bag.The_Items(Index).The_Item = The_Item then
                return True;
            end if;
        end loop;
        return False;
    end Is_A_Member;

    function Is_A_Subset (Left  : in Bag;
                          Right : in Bag) return Boolean is
        Right_Index : Natural;
    begin
        for Left_Index in 1 .. Left.The_Back loop
            Right_Index := Right.The_Back;
            while Right_Index > 0 loop
                if Left.The_Items(Left_Index).The_Item =
                   Right.The_Items(Right_Index).The_Item then
                    exit;
                else
                    Right_Index := Right_Index - 1;
                end if;
            end loop;
            if Right_Index = 0 then
                return False;
            elsif Left.The_Items(Left_Index).The_Count >
                  Right.The_Items(Right_Index).The_Count then
                return False;
            end if;
        end loop;
        return True;
    end Is_A_Subset;

    function Is_A_Proper_Subset (Left  : in Bag;
                                 Right : in Bag) return Boolean is
        Total_Left_Count  : Natural := 0;
        Total_Right_Count : Natural := 0;
        Right_Index       : Natural;
    begin
        for Left_Index in 1 .. Left.The_Back loop
            Right_Index := Right.The_Back;
            while Right_Index > 0 loop
                if Left.The_Items(Left_Index).The_Item =
                   Right.The_Items(Right_Index).The_Item then
                    exit;
                else
                    Right_Index := Right_Index - 1;
                end if;
            end loop;
            if Right_Index = 0 then
                return False;
            elsif Left.The_Items(Left_Index).The_Count >
                  Right.The_Items(Right_Index).The_Count then
                return False;
            end if;
            Total_Left_Count := Total_Left_Count +
                                Left.The_Items(Left_Index).The_Count;
        end loop;
        for Index in 1 .. Right.The_Back loop
            Total_Right_Count := Total_Right_Count +
                                 Right.The_Items(Index).The_Count;
        end loop;
        if Left.The_Back < Right.The_Back then
            return True;
        elsif Left.The_Back > Right.The_Back then
            return False;
        else
            return (Total_Left_Count < Total_Right_Count);
        end if;
    end Is_A_Proper_Subset;

end Bag_Simple_Sequential_Bounded_Managed_Noniterator;
```

BAG SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Bounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Bag : Bag
    OUTPUT
      From_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Unique_Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

END
IMPLEMENTATION ADA Bag_Simple_Sequential_Bounded_Managed_Noniterator
END
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Bag_Simple_Sequential_Unbounded_Managed_Iterator is

    type Bag is limited private;

    procedure Copy          (From_The_Bag : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Clear         (The_Bag      : in out Bag);
    procedure Add           (The_Item     : in      Item;
                             To_The_Bag   : in out Bag);
    procedure Remove        (The_Item     : in      Item;
                             From_The_Bag : in out Bag);
    procedure Union        (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Intersection (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Difference    (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  7 April 1995
--  adding procedures to replace functions

    procedure Is_Equal           (Left         : in Bag;
                                  Right        : in Bag;
                                  Result       : out Boolean);
    procedure Extent_Of          (The_Bag      : in Bag;
                                  Result       : out Natural);
    procedure Unique_Extent_Of   (The_Bag      : in Bag;
                                  Result       : out Natural);
    procedure Is_Empty           (The_Bag      : in Bag;
                                  Result       : out Boolean);
    procedure Is_A_Member        (The_Item     : in Item;
                                  Of_The_Bag   : in Bag;
                                  Result       : out Boolean);
    procedure Is_A_Subset        (Left         : in Bag;
                                  Right        : in Bag;
                                  Result       : out Boolean);
    procedure Is_A_Proper_Subset (Left         : in Bag;
                                  Right        : in Bag;
                                  Result       : out Boolean);

--  end of modification

    function Is_Equal           (Left         : in Bag;
                                 Right        : in Bag)  return Boolean;
    function Extent_Of          (The_Bag      : in Bag)  return Natural;
    function Unique_Extent_Of   (The_Bag      : in Bag)  return Natural;
    function Number_Of          (The_Item     : in Item;
                                 In_The_Bag   : in Bag)  return
Positive;
    function Is_Empty           (The_Bag      : in Bag)  return Boolean;
    function Is_A_Member        (The_Item     : in Item;
                                 Of_The_Bag   : in Bag)  return Boolean;
    function Is_A_Subset        (Left         : in Bag;
                                 Right        : in Bag)  return Boolean;
    function Is_A_Proper_Subset (Left         : in Bag;
                                 Right        : in Bag)  return Boolean;

    generic
        with procedure Process (The_Item  : in  Item;
                                The_Count : in  Positive;
                                Continue  : out Boolean);
    procedure Iterate (Over_The_Bag : in Bag);

    Overflow           : exception;
    Item_Is_Not_In_Bag : exception;

private
    type Node;
    type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Managed_Iterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--           "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body Bag_Simple_Sequential_Unbounded_Managed_Iterator is

    type Node is
        record
            The_Item  : Item;
            The_Count : Positive;
            Next      : Bag;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        The_Node.The_Count := 1;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Bag) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Bag is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                  (Item        => Node,
                                   Pointer     => Bag,
                                   Free        => Free,
                                   Set_Pointer => Set_Next,
                                   Pointer_Of  => Next_Of);

    procedure Copy (From_The_Bag : in     Bag;
                    To_The_Bag   : in out Bag) is
        From_Index : Bag := From_The_Bag;
        To_Index   : Bag;
    begin
        Node_Manager.Free(To_The_Bag);
        if From_The_Bag /= null then
            To_The_Bag := Node_Manager.New_Item;
            To_The_Bag.The_Item := From_Index.The_Item;
            To_The_Bag.The_Count := From_Index.The_Count;
            To_Index := To_The_Bag;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index := To_Index.Next;
                To_Index.The_Item := From_Index.The_Item;
                To_Index.The_Count := From_Index.The_Count;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Bag : in out Bag) is
    begin
        Node_Manager.Free(The_Bag);
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Bag : in out Bag) is
        Temporary_Node : Bag;
        Index          : Bag := To_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                Index.The_Count := Index.The_Count + 1;
                return;
            else
                Index := Index.Next;
            end if;
        end loop;
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.The_Count := 1;
        Temporary_Node.Next := To_The_Bag;
        To_The_Bag := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Bag : in out Bag) is
        Previous : Bag;
        Index    : Bag := From_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Index.The_Count > 1 then
                    Index.The_Count := Index.The_Count - 1;
                elsif Previous = null then
                    From_The_Bag := From_The_Bag.Next;
                    Index.Next := null;
                    Node_Manager.Free(Index);
                else
                    Previous.Next := Index.Next;
                    Index.Next := null;
                    Node_Manager.Free(Index);
                end if;
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Remove;

    procedure Union (Of_The_Bag : in     Bag;
                     And_The_Bag: in     Bag;
                     To_The_Bag : in out Bag) is
        From_Index     : Bag := Of_The_Bag;
        To_Index       : Bag;
        To_Top         : Bag;
        Temporary_Node : Bag;
    begin
        Node_Manager.Free(To_The_Bag);
        while From_Index /= null loop
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := From_Index.The_Item;
            Temporary_Node.The_Count := From_Index.The_Count;
            Temporary_Node.Next := To_The_Bag;
            To_The_Bag := Temporary_Node;
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Bag;
        To_Top := To_The_Bag;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                Temporary_Node := Node_Manager.New_Item;
                Temporary_Node.The_Item := From_Index.The_Item;
                Temporary_Node.The_Count := From_Index.The_Count;
                Temporary_Node.Next := To_The_Bag;
                To_The_Bag := Temporary_Node;
            else
                To_Index.The_Count :=
                  To_Index.The_Count + From_Index.The_Count;
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Bag  : in     Bag;
                            And_The_Bag : in     Bag;
                            To_The_Bag  : in out Bag) is
        Of_Index       : Bag := Of_The_Bag;
        And_Index      : Bag;
        Temporary_Node : Bag;
    begin
        Node_Manager.Free(To_The_Bag);
        while Of_Index /= null loop
            And_Index := And_The_Bag;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    Temporary_Node := Node_Manager.New_Item;
                    Temporary_Node.The_Item := Of_Index.The_Item;
                    if Of_Index.The_Count < And_Index.The_Count then
                        Temporary_Node.The_Count :=
Of_Index.The_Count;
                    else
                        Temporary_Node.The_Count :=
And_Index.The_Count;
                    end if;
                    Temporary_Node.Next := To_The_Bag;
                    To_The_Bag := Temporary_Node;
                    exit;
```

```
            else
                And_Index := And_Index.Next;
            end if;
        end loop;
        Of_Index := Of_Index.Next;
    end loop;
exception
    when Storage_Error =>
        raise Overflow;
end Intersection;

procedure Difference (Of_The_Bag  : in     Bag;
                      And_The_Bag : in     Bag;
                      To_The_Bag  : in out Bag) is
    Of_Index       : Bag := Of_The_Bag;
    And_Index      : Bag;
    Temporary_Node : Bag;
begin
    Node_Manager.Free(To_The_Bag);
    while Of_Index /= null loop
        And_Index := And_The_Bag;
        while And_Index /= null loop
            if Of_Index.The_Item = And_Index.The_Item then
                exit;
            else
                And_Index := And_Index.Next;
            end if;
        end loop;
        if And_Index = null then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := Of_Index.The_Item;
            Temporary_Node.The_Count := Of_Index.The_Count;
            Temporary_Node.Next := To_The_Bag;
            To_The_Bag := Temporary_Node;
        elsif Of_Index.The_Count > And_Index.The_Count then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := Of_Index.The_Item;
            Temporary_Node.The_Count := Of_Index.The_Count -
                                        And_Index.The_Count;
            Temporary_Node.Next := To_The_Bag;
            To_The_Bag := Temporary_Node;
        end if;
        Of_Index := Of_Index.Next;
    end loop;
exception
    when Storage_Error =>
        raise Overflow;
end Difference;

-- modified by Tuan Nguyen and Vincent Hong
-- date:  8 April 1995
-- adding procedures to replace functions

procedure Is_Equal (Left   : in Bag;
                    Right  : in Bag;
                    Result : out Boolean) is
begin
    Result := Is_Equal(Left,Right);
end Is_Equal;

procedure Extent_Of (The_Bag : in Bag;
                     Result  : out Natural) is
begin
    Result := Extent_Of(The_Bag);
end Extent_Of;

procedure Unique_Extent_Of (The_Bag : in Bag;
                            Result  : out Natural) is
begin
    Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;

procedure Number_Of (The_Item   : in Item;
                     In_The_Bag : in Bag;
                     Result     : out Positive) is
begin
        Result := Number_Of(The_Item,In_The_Bag);
end Number_Of;

procedure Is_Empty (The_Bag : in Bag;
                    Result  : out Boolean) is
begin
    Result := Is_Empty(The_Bag);
end Is_Empty;

procedure Is_A_Member (The_Item   : in Item;
                       Of_The_Bag : in Bag;
                       Result     : out Boolean) is
begin
    Result := Is_A_Member(The_Item,Of_The_Bag);
end Is_A_Member;

procedure Is_A_Subset (Left   : in Bag;
                       Right  : in Bag;
                       Result : out Boolean) is
begin
    Result := Is_A_Subset(Left,Right);
end Is_A_Subset;

procedure Is_A_Proper_Subset (Left   : in Bag;
                              Right  : in Bag;
                              Result : out Boolean) is
begin
    Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;

-- end of modification.

function Is_Equal (Left  : in Bag;
                   Right : in Bag) return Boolean is
    Left_Count  : Natural := 0;
    Right_Count : Natural := 0;
    Left_Index  : Bag     := Left;
    Right_Index : Bag;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        if Right_Index = null then
            return False;
        elsif Left_Index.The_Count /= Right_Index.The_Count then
            return False;
        else
            Left_Count := Left_Count + 1;
            Left_Index := Left_Index.Next;
        end if;
    end loop;
    Right_Index := Right;
    while Right_Index /= null loop
        Right_Count := Right_Count + 1;
        Right_Index := Right_Index.Next;
    end loop;
    return (Left_Count = Right_Count);
end Is_Equal;

function Extent_Of (The_Bag : in Bag) return Natural is
    Count : Natural := 0;
    Index : Bag     := The_Bag;
begin
    while Index /= null loop
        Count := Count + Index.The_Count;
        Index := Index.Next;
    end loop;
    return Count;
end Extent_Of;

function Unique_Extent_Of (The_Bag : in Bag) return Natural is
    Count : Natural := 0;
    Index : Bag     := The_Bag;
begin
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Unique_Extent_Of;

function Number_Of (The_Item   : in Item;
                    In_The_Bag : in Bag) return Positive is
    Index : Bag := In_The_Bag;
begin
    while Index /= null loop
        if The_Item = Index.The_Item then
            return Index.The_Count;
        else
            Index := Index.Next;
        end if;
    end loop;
    raise Item_Is_Not_In_Bag;
end Number_Of;

function Is_Empty (The_Bag : in Bag) return Boolean is
begin
    return (The_Bag = null);
end Is_Empty;

function Is_A_Member (The_Item   : in Item;
                      Of_The_Bag : in Bag) return Boolean is
    Index : Bag := Of_The_Bag;
begin
    while Index /= null loop
        if The_Item = Index.The_Item then
            return True;
        end if;
        Index := Index.Next;
    end loop;
    return False;
end Is_A_Member;

function Is_A_Subset (Left  : in Bag;
                      Right : in Bag) return Boolean is
    Left_Index  : Bag := Left;
    Right_Index : Bag;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        if Right_Index = null then
            return False;
        elsif Left_Index.The_Count > Right_Index.The_Count then
            return False;
        else
            Left_Index := Left_Index.Next;
        end if;
```

```
        end loop;
        return True;
    end Is_A_Subset;

    function Is_A_Proper_Subset (Left  : in Bag;
                                 Right : in Bag) return Boolean is
        Unique_Left_Count  : Natural := 0;
        Unique_Right_Count : Natural := 0;
        Total_Left_Count   : Natural := 0;
        Total_Right_Count  : Natural := 0;
        Left_Index         : Bag     := Left;
        Right_Index        : Bag;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            elsif Left_Index.The_Count > Right_Index.The_Count then
                return False;
            else
                Unique_Left_Count := Unique_Left_Count + 1;
                Total_Left_Count := Total_Left_Count +
Left_Index.The_Count;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Unique_Right_Count := Unique_Right_Count + 1;
            Total_Right_Count := Total_Right_Count +
Right_Index.The_Count;
            Right_Index := Right_Index.Next;
        end loop;
        if Unique_Left_Count < Unique_Right_Count then
            return True;
        elsif Unique_Left_Count > Unique_Right_Count then
            return False;
        else
            return (Total_Left_Count < Total_Right_Count);
        end if;
    end Is_A_Proper_Subset;

    procedure Iterate (Over_The_Bag : in Bag) is
        The_Iterator : Bag      := Over_The_Bag;
        Continue     : Boolean;
    begin
        while The_Iterator /= null loop
            Process(The_Iterator.The_Item, The_Iterator.The_Count,
Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Bag_Simple_Sequential_Unbounded_Managed_Iterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Bag : Bag
    OUTPUT
      From_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Unique_Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], The_Count : in[t :
Positive], Continue : out[t : Boolean]]
    INPUT
      Over_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

END
IMPLEMENTATION ADA Bag_Simple_Sequential_Unbounded_Managed_Iterator
END
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
   type Item is private;
package Bag_Simple_Sequential_Unbounded_Managed_Noniterator is

    type Bag is limited private;

    procedure Copy          (From_The_Bag : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Clear         (The_Bag      : in out Bag);
    procedure Add           (The_Item     : in      Item;
                             To_The_Bag   : in out Bag);
    procedure Remove        (The_Item     : in      Item;
                             From_The_Bag : in out Bag);
    procedure Union         (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Intersection  (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Difference    (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  7 April 1995
--  adding procedures to replace functions

    procedure Is_Equal            (Left        : in Bag;
                                   Right       : in Bag;
                                   Result      : out Boolean);
    procedure Extent_Of           (The_Bag     : in Bag;
                                   Result      : out Natural);
    procedure Unique_Extent_Of    (The_Bag     : in Bag;
                                   Result      : out Natural);
    procedure Is_Empty            (The_Bag     : in Bag;
                                   Result      : out Boolean);
    procedure Is_A_Member         (The_Item    : in Item;
                                   Of_The_Bag  : in Bag;
                                   Result      : out Boolean);
    procedure Is_A_Subset         (Left        : in Bag;
                                   Right       : in Bag;
                                   Result      : out Boolean);
    procedure Is_A_Proper_Subset  (Left        : in Bag;
                                   Right       : in Bag;
                                   Result      : out Boolean);

--  end of modification

    function Is_Equal           (Left       : in Bag;
                                 Right      : in Bag)  return Boolean;
    function Extent_Of          (The_Bag    : in Bag)  return Natural;
    function Unique_Extent_Of   (The_Bag    : in Bag)  return Natural;
    function Number_Of          (The_Item   : in Item;
                                 In_The_Bag : in Bag)  return
Positive;
    function Is_Empty           (The_Bag    : in Bag)  return Boolean;
    function Is_A_Member        (The_Item   : in Item;
                                 Of_The_Bag : in Bag)  return Boolean;
    function Is_A_Subset        (Left       : in Bag;
                                 Right      : in Bag)  return Boolean;
    function Is_A_Proper_Subset (Left       : in Bag;
                                 Right      : in Bag)  return Boolean;

    Overflow          : exception;
    Item_Is_Not_In_Bag : exception;

private
    type Node;
    type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Managed_Noniterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--           "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body Bag_Simple_Sequential_Unbounded_Managed_Noniterator is

    type Node is
        record
            The_Item  : Item;
            The_Count : Positive;
            Next      : Bag;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        The_Node.The_Count := 1;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Bag) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Bag is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Bag,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Copy (From_The_Bag : in     Bag;
                    To_The_Bag   : in out Bag) is
        From_Index : Bag := From_The_Bag;
        To_Index   : Bag;
    begin
        Node_Manager.Free(To_The_Bag);
        if From_The_Bag /= null then
            To_The_Bag := Node_Manager.New_Item;
            To_The_Bag.The_Item := From_Index.The_Item;
            To_The_Bag.The_Count := From_Index.The_Count;
            To_Index := To_The_Bag;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index := To_Index.Next;
                To_Index.The_Item := From_Index.The_Item;
                To_Index.The_Count := From_Index.The_Count;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Bag : in out Bag) is
    begin
        Node_Manager.Free(The_Bag);
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Bag : in out Bag) is
        Temporary_Node : Bag;
        Index          : Bag := To_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                Index.The_Count := Index.The_Count + 1;
                return;
            else
                Index := Index.Next;
            end if;
        end loop;
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.The_Count := 1;
        Temporary_Node.Next := To_The_Bag;
        To_The_Bag := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Bag : in out Bag) is
        Previous : Bag;
        Index    : Bag := From_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Index.The_Count > 1 then
                    Index.The_Count := Index.The_Count - 1;
                elsif Previous = null then
                    From_The_Bag := From_The_Bag.Next;
                    Index.Next := null;
                    Node_Manager.Free(Index);
                else
                    Previous.Next := Index.Next;
                    Index.Next := null;
                    Node_Manager.Free(Index);
                end if;
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Remove;

    procedure Union (Of_The_Bag : in     Bag;
                     And_The_Bag: in     Bag;
                     To_The_Bag : in out Bag) is
        From_Index     : Bag := Of_The_Bag;
        To_Index       : Bag;
        To_Top         : Bag;
        Temporary_Node : Bag;
    begin
        Node_Manager.Free(To_The_Bag);
        while From_Index /= null loop
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := From_Index.The_Item;
            Temporary_Node.The_Count := From_Index.The_Count;
            Temporary_Node.Next := To_The_Bag;
            To_The_Bag := Temporary_Node;
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Bag;
        To_Top := To_The_Bag;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                Temporary_Node := Node_Manager.New_Item;
                Temporary_Node.The_Item := From_Index.The_Item;
                Temporary_Node.The_Count := From_Index.The_Count;
                Temporary_Node.Next := To_The_Bag;
                To_The_Bag := Temporary_Node;
            else
                To_Index.The_Count :=
                  To_Index.The_Count + From_Index.The_Count;
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Bag  : in     Bag;
                            And_The_Bag : in     Bag;
                            To_The_Bag  : in out Bag) is
        Of_Index       : Bag := Of_The_Bag;
        And_Index      : Bag;
        Temporary_Node : Bag;
    begin
        Node_Manager.Free(To_The_Bag);
        while Of_Index /= null loop
            And_Index := And_The_Bag;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    Temporary_Node := Node_Manager.New_Item;
                    Temporary_Node.The_Item := Of_Index.The_Item;
                    if Of_Index.The_Count < And_Index.The_Count then
                        Temporary_Node.The_Count :=
Of_Index.The_Count;
                    else
                        Temporary_Node.The_Count :=
And_Index.The_Count;
                    end if;
                    Temporary_Node.Next := To_The_Bag;
                    To_The_Bag := Temporary_Node;
                    exit;
```

```
            else
                And_Index := And_Index.Next;
            end if;
        end loop;
        Of_Index := Of_Index.Next;
    end loop;
exception
    when Storage_Error =>
        raise Overflow;
end Intersection;

procedure Difference (Of_The_Bag  : in     Bag;
                      And_The_Bag : in     Bag;
                      To_The_Bag  : in out Bag) is
    Of_Index       : Bag := Of_The_Bag;
    And_Index      : Bag;
    Temporary_Node : Bag;
begin
    Node_Manager.Free(To_The_Bag);
    while Of_Index /= null loop
        And_Index := And_The_Bag;
        while And_Index /= null loop
            if Of_Index.The_Item = And_Index.The_Item then
                exit;
            else
                And_Index := And_Index.Next;
            end if;
        end loop;
        if And_Index = null then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := Of_Index.The_Item;
            Temporary_Node.The_Count := Of_Index.The_Count;
            Temporary_Node.Next := To_The_Bag;
            To_The_Bag := Temporary_Node;
        elsif Of_Index.The_Count > And_Index.The_Count then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := Of_Index.The_Item;
            Temporary_Node.The_Count := Of_Index.The_Count -
                                        And_Index.The_Count;
            Temporary_Node.Next := To_The_Bag;
            To_The_Bag := Temporary_Node;
        end if;
        Of_Index := Of_Index.Next;
    end loop;
exception
    when Storage_Error =>
        raise Overflow;
end Difference;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

procedure Is_Equal (Left   : in Bag;
                    Right  : in Bag;
                    Result : out Boolean) is
begin
    Result := Is_Equal(Left,Right);
end Is_Equal;

procedure Extent_Of (The_Bag : in Bag;
                     Result  : out Natural) is
begin
    Result := Extent_Of(The_Bag);
end Extent_Of;

procedure Unique_Extent_Of (The_Bag : in Bag;
                            Result  : out Natural) is
begin
    Result := Unique_Extent_Of (The_Bag);
end Unique_Extent_Of;

procedure Number_Of (The_Item   : in Item;
                     In_The_Bag : in Bag;
                     Result     : out Positive) is
begin
        Result := Number_Of(The_Item,In_The_Bag);
end Number_Of;

procedure Is_Empty (The_Bag : in Bag;
                    Result  : out Boolean) is
begin
    Result := Is_Empty(The_Bag);
end Is_Empty;

procedure Is_A_Member (The_Item   : in Item;
                       Of_The_Bag : in Bag;
                       Result     : out Boolean) is
begin
    Result := Is_A_Member(The_Item,Of_The_Bag);
end Is_A_Member;

procedure Is_A_Subset (Left   : in Bag;
                       Right  : in Bag;
                       Result : out Boolean) is
begin
    Result := Is_A_Subset(Left,Right);
end Is_A_Subset;

procedure Is_A_Proper_Subset (Left   : in Bag;
                              Right  : in Bag;
                              Result : out Boolean) is
begin
    Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;

--  end of modification

function Is_Equal (Left  : in Bag;
                   Right : in Bag) return Boolean is
    Left_Count  : Natural := 0;
    Right_Count : Natural := 0;
    Left_Index  : Bag     := Left;
    Right_Index : Bag;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
           end if;
        end loop;
        if Right_Index = null then
            return False;
        elsif Left_Index.The_Count /= Right_Index.The_Count then
            return False;
        else
            Left_Count := Left_Count + 1;
            Left_Index := Left_Index.Next;
        end if;
    end loop;
    Right_Index := Right;
    while Right_Index /= null loop
        Right_Count := Right_Count + 1;
        Right_Index := Right_Index.Next;
    end loop;
    return (Left_Count = Right_Count);
end Is_Equal;

function Extent_Of (The_Bag : in Bag) return Natural is
    Count : Natural := 0;
    Index : Bag     := The_Bag;
begin
    while Index /= null loop
        Count := Count + Index.The_Count;
        Index := Index.Next;
    end loop;
    return Count;
end Extent_Of;

function Unique_Extent_Of (The_Bag : in Bag) return Natural is
    Count : Natural := 0;
    Index : Bag     := The_Bag;
begin
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Unique_Extent_Of;

function Number_Of (The_Item   : in Item;
                    In_The_Bag : in Bag) return Positive is
    Index : Bag := In_The_Bag;
begin
    while Index /= null loop
        if The_Item = Index.The_Item then
            return Index.The_Count;
        else
            Index := Index.Next;
        end if;
    end loop;
    raise Item_Is_Not_In_Bag;
end Number_Of;

function Is_Empty (The_Bag : in Bag) return Boolean is
begin
    return (The_Bag = null);
end Is_Empty;

function Is_A_Member (The_Item   : in Item;
                      Of_The_Bag : in Bag) return Boolean is
    Index : Bag := Of_The_Bag;
begin
    while Index /= null loop
        if The_Item = Index.The_Item then
            return True;
        end if;
        Index := Index.Next;
    end loop;
    return False;
end Is_A_Member;

function Is_A_Subset (Left  : in Bag;
                      Right : in Bag) return Boolean is
    Left_Index  : Bag := Left;
    Right_Index : Bag;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        if Right_Index = null then
            return False;
        elsif Left_Index.The_Count > Right_Index.The_Count then
            return False;
        else
            Left_Index := Left_Index.Next;
        end if;
```

```
    end loop;
    return True;
end Is_A_Subset;

function Is_A_Proper_Subset (Left  : in Bag;
                             Right : in Bag) return Boolean is
    Unique_Left_Count  : Natural := 0;
    Unique_Right_Count : Natural := 0;
    Total_Left_Count   : Natural := 0;
    Total_Right_Count  : Natural := 0;
    Left_Index         : Bag     := Left;
    Right_Index        : Bag;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        if Right_Index = null then
            return False;
        elsif Left_Index.The_Count > Right_Index.The_Count then
                return False;
            else
                Unique_Left_Count := Unique_Left_Count + 1;
                Total_Left_Count := Total_Left_Count +
Left_Index.The_Count;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Unique_Right_Count := Unique_Right_Count + 1;
            Total_Right_Count := Total_Right_Count +
Right_Index.The_Count;
            Right_Index := Right_Index.Next;
        end loop;
        if Unique_Left_Count < Unique_Right_Count then
            return True;
        elsif Unique_Left_Count > Unique_Right_Count then
            return False;
        else
            return (Total_Left_Count < Total_Right_Count);
        end if;
    end Is_A_Proper_Subset;

end Bag_Simple_Sequential_Unbounded_Managed_Noniterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Bag : Bag
    OUTPUT
      From_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Unique_Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

END
IMPLEMENTATION ADA Bag_Simple_Sequential_Unbounded_Managed_Noniterator
END
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator is

    type Bag is limited private;

    procedure Copy          (From_The_Bag : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Clear         (The_Bag      : in out Bag);
    procedure Add           (The_Item     : in      Item;
                             To_The_Bag   : in out Bag);
    procedure Remove        (The_Item     : in      Item;
                             From_The_Bag : in out Bag);
    procedure Union         (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                            To_The_Bag    : in out Bag);
    procedure Intersection (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);
    procedure Difference    (Of_The_Bag   : in      Bag;
                             And_The_Bag  : in      Bag;
                             To_The_Bag   : in out Bag);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  7 April 1995
--  adding procedures to replace functions

    procedure Is_Equal              (Left         : in Bag;
                                     Right        : in Bag;
                                     Result       : out Boolean);
    procedure Extent_Of             (The_Bag      : in Bag;
                                     Result       : out Natural);
    procedure Unique_Extent_Of      (The_Bag      : in Bag;
                                     Result       : out Natural);
    procedure Is_Empty              (The_Bag      : in Bag;
                                     Result       : out Boolean);
    procedure Is_A_Member           (The_Item     : in Item;
                                     Of_The_Bag : in Bag;
```

```
                                     Result      : out Boolean);
    procedure Is_A_Subset           (Left        : in Bag;
                                     Right       : in Bag;
                                     Result      : out Boolean);
    procedure Is_A_Proper_Subset (Left        : in Bag;
                                     Right       : in Bag;
                                     Result      : out Boolean);

--  end of modification

    function Is_Equal              (Left        : in Bag;
                                    Right       : in Bag)  return Boolean;
    function Extent_Of             (The_Bag     : in Bag)  return Natural;
    function Unique_Extent_Of      (The_Bag     : in Bag)  return Natural;
    function Number_Of             (The_Item    : in Item;
                                    In_The_Bag : in Bag)  return
Positive;
    function Is_Empty              (The_Bag     : in Bag)  return Boolean;
    function Is_A_Member           (The_Item    : in Item;
                                    Of_The_Bag : in Bag)  return Boolean;
    function Is_A_Subset           (Left        : in Bag;
                                    Right       : in Bag)  return Boolean;
    function Is_A_Proper_Subset (Left        : in Bag;
                                    Right       : in Bag)  return Boolean;

    generic
        with procedure Process (The_Item  : in  Item;
                                The_Count : in  Positive;
                                Continue  : out Boolean);
    procedure Iterate (Over_The_Bag : in Bag);

    Overflow          : exception;
    Item_Is_Not_In_Bag : exception;

private
    type Node;
    type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--           "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator is

    type Node is
        record
            The_Item  : Item;
            The_Count : Positive;
            Next      : Bag;
        end record;

    procedure Copy (From_The_Bag : in     Bag;
                    To_The_Bag   : in out Bag) is
        From_Index : Bag := From_The_Bag;
        To_Index   : Bag;
    begin
        if From_The_Bag = null then
            To_The_Bag := null;
        else
            To_The_Bag := new Node'(The_Item  => From_Index.The_Item,
                                    The_Count => From_Index.The_Count,
                                    Next      => null);
            To_Index := To_The_Bag;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item  =>
From_Index.The_Item,
                                           The_Count =>
From_Index.The_Count,
                                           Next      => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Bag : in out Bag) is
    begin
        The_Bag := null;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Bag : in out Bag) is
        Index : Bag := To_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                Index.The_Count := Index.The_Count + 1;
                return;
            else
                Index := Index.Next;
            end if;
        end loop;
        To_The_Bag := new Node'(The_Item  => The_Item,
                                The_Count => 1,
                                Next      => To_The_Bag);
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Bag : in out Bag) is
        Previous : Bag;
        Index    : Bag := From_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Index.The_Count > 1 then
                    Index.The_Count := Index.The_Count - 1;
                elsif Previous = null then
                    From_The_Bag := From_The_Bag.Next;
                else
                    Previous.Next := Index.Next;
                end if;
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Remove;

    procedure Union (Of_The_Bag  : in     Bag;
                     And_The_Bag: in     Bag;
                     To_The_Bag  : in out Bag) is
        From_Index : Bag := Of_The_Bag;
        To_Index   : Bag;
        To_Top     : Bag;
    begin
        To_The_Bag := null;
        while From_Index /= null loop
            To_The_Bag := new Node'(The_Item  => From_Index.The_Item,
                                    The_Count => From_Index.The_Count,
                                    Next      => To_The_Bag);
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Bag;
        To_Top := To_The_Bag;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                To_The_Bag := new Node'(The_Item  =>
From_Index.The_Item,
                                        The_Count =>
From_Index.The_Count,
                                        Next      => To_The_Bag);
            else
                To_Index.The_Count :=
                  To_Index.The_Count + From_Index.The_Count;
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Bag  : in     Bag;
                            And_The_Bag : in     Bag;
                            To_The_Bag  : in out Bag) is
        Of_Index  : Bag := Of_The_Bag;
        And_Index : Bag;
    begin
        To_The_Bag := null;
        while Of_Index /= null loop
            And_Index := And_The_Bag;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    if Of_Index.The_Count < And_Index.The_Count then
                        To_The_Bag :=
                          new Node'(The_Item  => Of_Index.The_Item,
                                    The_Count => Of_Index.The_Count,
                                    Next      => To_The_Bag);
                    else
                        To_The_Bag :=
                          new Node'(The_Item  => And_Index.The_Item,
                                    The_Count => And_Index.The_Count,
                                    Next      => To_The_Bag);
                    end if;
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Bag  : in     Bag;
                          And_The_Bag : in     Bag;
                          To_The_Bag  : in out Bag) is
        Of_Index  : Bag := Of_The_Bag;
        And_Index : Bag;
    begin
        To_The_Bag:= null;
        while Of_Index /= null loop
            And_Index := And_The_Bag;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            if And_Index = null then
                To_The_Bag := new Node'(The_Item  =>
Of_Index.The_Item,
                                        The_Count =>
Of_Index.The_Count,
```

```
                                            Next       => To_The_Bag);
          elsif Of_Index.The_Count > And_Index.The_Count then
              To_The_Bag := new Node'(The_Item  =>
Of_Index.The_Item,
                                            The_Count =>
Of_Index.The_Count -
And_Index.The_Count,
                                            Next      => To_The_Bag);
          end if;
          Of_Index := Of_Index.Next;
      end loop;
   exception
      when Storage_Error =>
          raise Overflow;
   end Difference;

--   modified by Tuan Nguyen and Vincent Hong
--   date:  8 April 1995
--   adding procedures to replace functions

   procedure Is_Equal (Left   : in Bag;
                       Right  : in Bag;
                       Result : out Boolean) is
   begin
      Result := Is_Equal(Left,Right);
   end Is_Equal;

   procedure Extent_Of (The_Bag : in Bag;
                        Result  : out Natural) is
   begin
      Result := Extent_Of(The_Bag);
   end Extent_Of;

   procedure Unique_Extent_Of (The_Bag : in Bag;
                               Result  : out Natural) is
   begin
      Result := Unique_Extent_Of (The_Bag);
   end Unique_Extent_Of;

   procedure Number_Of (The_Item   : in Item;
                        In_The_Bag : in Bag;
                        Result     : out Positive) is
   begin
          Result := Number_Of(The_Item,In_The_Bag);
   end Number_Of;

   procedure Is_Empty (The_Bag : in Bag;
                       Result  : out Boolean) is
   begin
      Result := Is_Empty(The_Bag);
   end Is_Empty;

   procedure Is_A_Member (The_Item   : in Item;
                          Of_The_Bag : in Bag;
                          Result     : out Boolean) is
   begin
      Result := Is_A_Member(The_Item,Of_The_Bag);
   end Is_A_Member;

   procedure Is_A_Subset (Left   : in Bag;
                          Right  : in Bag;
                          Result : out Boolean) is
   begin
      Result := Is_A_Subset(Left,Right);
   end Is_A_Subset;

   procedure Is_A_Proper_Subset (Left   : in Bag;
                                 Right  : in Bag;
                                 Result : out Boolean) is
   begin
      Result := Is_A_Proper_Subset(Left,Right);
   end Is_A_Proper_Subset;

--   end of modification

   function Is_Equal (Left  : in Bag;
                      Right : in Bag) return Boolean is
      Left_Count  : Natural := 0;
      Right_Count : Natural := 0;
      Left_Index  : Bag     := Left;
      Right_Index : Bag;
   begin
      while Left_Index /= null loop
          Right_Index := Right;
          while Right_Index /= null loop
              if Left_Index.The_Item = Right_Index.The_Item then
                  exit;
              else
                  Right_Index := Right_Index.Next;
             end if;
          end loop;
          if Right_Index = null then
              return False;
          elsif Left_Index.The_Count /= Right_Index.The_Count then
              return False;
          else
              Left_Count := Left_Count + 1;
              Left_Index := Left_Index.Next;
          end if;
      end loop;
      Right_Index := Right;
      while Right_Index /= null loop
          Right_Count := Right_Count + 1;
          Right_Index := Right_Index.Next;
      end loop;
      return (Left_Count = Right_Count);
   end Is_Equal;

   function Extent_Of (The_Bag : in Bag) return Natural is
      Count : Natural := 0;
      Index : Bag     := The_Bag;
   begin
      while Index /= null loop
          Count := Count + Index.The_Count;
          Index := Index.Next;
      end loop;
      return Count;
   end Extent_Of;

   function Unique_Extent_Of (The_Bag : in Bag) return Natural is
      Count : Natural := 0;
      Index : Bag     := The_Bag;
   begin
      while Index /= null loop
          Count := Count + 1;
          Index := Index.Next;
      end loop;
      return Count;
   end Unique_Extent_Of;

   function Number_Of (The_Item   : in Item;
                       In_The_Bag : in Bag) return Positive is
      Index : Bag := In_The_Bag;
   begin
      while Index /= null loop
          if The_Item = Index.The_Item then
              return Index.The_Count;
          else
              Index := Index.Next;
          end if;
      end loop;
      raise Item_Is_Not_In_Bag;
   end Number_Of;

   function Is_Empty (The_Bag : in Bag) return Boolean is
   begin
      return (The_Bag = null);
   end Is_Empty;

   function Is_A_Member (The_Item   : in Item;
                         Of_The_Bag : in Bag) return Boolean is
      Index : Bag := Of_The_Bag;
   begin
      while Index /= null loop
          if The_Item = Index.The_Item then
              return True;
          end if;
          Index := Index.Next;
      end loop;
      return False;
   end Is_A_Member;

   function Is_A_Subset (Left  : in Bag;
                         Right : in Bag) return Boolean is
      Left_Index  : Bag := Left;
      Right_Index : Bag;
   begin
      while Left_Index /= null loop
          Right_Index := Right;
          while Right_Index /= null loop
              if Left_Index.The_Item = Right_Index.The_Item then
                  exit;
              else
                  Right_Index := Right_Index.Next;
              end if;
          end loop;
          if Right_Index = null then
              return False;
          elsif Left_Index.The_Count > Right_Index.The_Count then
              return False;
          else
              Left_Index := Left_Index.Next;
          end if;
      end loop;
      return True;
   end Is_A_Subset;

   function Is_A_Proper_Subset (Left  : in Bag;
                                Right : in Bag) return Boolean is
      Unique_Left_Count  : Natural := 0;
      Unique_Right_Count : Natural := 0;
      Total_Left_Count   : Natural := 0;
      Total_Right_Count  : Natural := 0;
      Left_Index         : Bag     := Left;
      Right_Index        : Bag;
   begin
      while Left_Index /= null loop
          Right_Index := Right;
          while Right_Index /= null loop
              if Left_Index.The_Item = Right_Index.The_Item then
                  exit;
              else
                  Right_Index := Right_Index.Next;
              end if;
          end loop;
          if Right_Index = null then
              return False;
          elsif Left_Index.The_Count > Right_Index.The_Count then
              return False;
          else
              Unique_Left_Count := Unique_Left_Count + 1;
              Total_Left_Count := Total_Left_Count +
Left_Index.The_Count;
              Left_Index := Left_Index.Next;
          end if;
```

```
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Unique_Right_Count := Unique_Right_Count + 1;
            Total_Right_Count := Total_Right_Count +
Right_Index.The_Count;
            Right_Index := Right_Index.Next;
        end loop;
        if Unique_Left_Count < Unique_Right_Count then
            return True;
        elsif Unique_Left_Count > Unique_Right_Count then
            return False;
        else
            return (Total_Left_Count < Total_Right_Count);
        end if;
    end Is_A_Proper_Subset;

    procedure Iterate (Over_The_Bag : in Bag) is
        The_Iterator : Bag      := Over_The_Bag;
        Continue     : Boolean;
    begin
        while The_Iterator /= null loop
            Process(The_Iterator.The_Item, The_Iterator.The_Count,
Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Bag : Bag
    OUTPUT
      From_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Unique_Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], The_Count : in[t :
Positive], Continue : out[t : Boolean]]
    INPUT
      Over_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

END
IMPLEMENTATION ADA Bag_Simple_Sequential_Unbounded_Unmanaged_Iterator
END
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator is

    type Bag is limited private;

    procedure Copy          (From_The_Bag : in      Bag;
                              To_The_Bag   : in out Bag);
    procedure Clear         (The_Bag      : in out Bag);
    procedure Add           (The_Item     : in      Item;
                              To_The_Bag   : in out Bag);
    procedure Remove        (The_Item     : in      Item;
                              From_The_Bag : in out Bag);
    procedure Union         (Of_The_Bag   : in      Bag;
                              And_The_Bag  : in      Bag;
                              To_The_Bag   : in out Bag);
    procedure Intersection  (Of_The_Bag   : in      Bag;
                              And_The_Bag  : in      Bag;
                              To_The_Bag   : in out Bag);
    procedure Difference    (Of_The_Bag   : in      Bag;
                              And_The_Bag  : in      Bag;
                              To_The_Bag   : in out Bag);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  7 April 1995
--  adding procedures to replace functions

    procedure Is_Equal              (Left        : in Bag;
                                     Right       : in Bag;
                                     Result      : out Boolean);
    procedure Extent_Of             (The_Bag     : in Bag;
                                     Result      : out Natural);
    procedure Unique_Extent_Of      (The_Bag     : in Bag;
                                     Result      : out Natural);
    procedure Is_Empty              (The_Bag     : in Bag;
                                     Result      : out Boolean);
    procedure Is_A_Member           (The_Item    : in Item;
                                     Of_The_Bag  : in Bag;
                                     Result      : out Boolean);
    procedure Is_A_Subset           (Left        : in Bag;
                                     Right       : in Bag;
                                     Result      : out Boolean);
    procedure Is_A_Proper_Subset    (Left        : in Bag;
                                     Right       : in Bag;
                                     Result      : out Boolean);

--  end of modification

    function Is_Equal           (Left        : in Bag;
                                 Right       : in Bag)  return Boolean;
    function Extent_Of          (The_Bag     : in Bag)  return Natural;
    function Unique_Extent_Of   (The_Bag     : in Bag)  return Natural;
    function Number_Of          (The_Item    : in Item;
                                 In_The_Bag  : in Bag)  return
Positive;
    function Is_Empty           (The_Bag     : in Bag)  return Boolean;
    function Is_A_Member        (The_Item    : in Item;
                                 Of_The_Bag  : in Bag)  return Boolean;
    function Is_A_Subset        (Left        : in Bag;
                                 Right       : in Bag)  return Boolean;
    function Is_A_Proper_Subset (Left        : in Bag;
                                 Right       : in Bag)  return Boolean;

    Overflow          : exception;
    Item_Is_Not_In_Bag : exception;

private
    type Node;
    type Bag is access Node;
end Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator is

    type Node is
        record
            The_Item  : Item;
            The_Count : Positive;
            Next      : Bag;
        end record;

    procedure Copy (From_The_Bag : in     Bag;
                    To_The_Bag   : in out Bag) is
        From_Index : Bag := From_The_Bag;
        To_Index   : Bag;
    begin
        if From_The_Bag = null then
            To_The_Bag := null;
        else
            To_The_Bag := new Node'(The_Item  => From_Index.The_Item,
                                    The_Count => From_Index.The_Count,
                                    Next      => null);
            To_Index := To_The_Bag;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item  =>
From_Index.The_Item,
                                           The_Count =>
From_Index.The_Count,
                                           Next      => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Bag : in out Bag) is
    begin
        The_Bag := null;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Bag : in out Bag) is
        Index : Bag := To_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                Index.The_Count := Index.The_Count + 1;
                return;
            else
                Index := Index.Next;
            end if;
        end loop;
        To_The_Bag := new Node'(The_Item  => The_Item,
                                The_Count => 1,
                                Next      => To_The_Bag);
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Bag : in out Bag) is
        Previous : Bag;
        Index    : Bag := From_The_Bag;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Index.The_Count > 1 then
                    Index.The_Count := Index.The_Count - 1;
                elsif Previous = null then
                    From_The_Bag := From_The_Bag.Next;
                else
                    Previous.Next := Index.Next;
                end if;
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Remove;

    procedure Union (Of_The_Bag : in     Bag;
                     And_The_Bag: in     Bag;
                     To_The_Bag : in out Bag) is
        From_Index : Bag := Of_The_Bag;
        To_Index   : Bag;
        To_Top     : Bag;
    begin
        To_The_Bag := null;
        while From_Index /= null loop
            To_The_Bag := new Node'(The_Item  => From_Index.The_Item,
                                    The_Count => From_Index.The_Count,
                                    Next      => To_The_Bag);
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Bag;
        To_Top := To_The_Bag;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                To_The_Bag := new Node'(The_Item  =>
From_Index.The_Item,
                                        The_Count =>
From_Index.The_Count,
                                        Next      => To_The_Bag);
            else
                To_Index.The_Count :=
                  To_Index.The_Count + From_Index.The_Count;
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Bag  : in     Bag;
                            And_The_Bag : in     Bag;
                            To_The_Bag  : in out Bag) is
        Of_Index  : Bag := Of_The_Bag;
        And_Index : Bag;
    begin
        To_The_Bag := null;
        while Of_Index /= null loop
            And_Index := And_The_Bag;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    if Of_Index.The_Count < And_Index.The_Count then
                        To_The_Bag :=
                          new Node'(The_Item  => Of_Index.The_Item,
                                    The_Count => Of_Index.The_Count,
                                    Next      => To_The_Bag);
                    else
                        To_The_Bag :=
                          new Node'(The_Item  => And_Index.The_Item,
                                    The_Count => And_Index.The_Count,
                                    Next      => To_The_Bag);
                    end if;
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Bag  : in     Bag;
                          And_The_Bag : in     Bag;
                          To_The_Bag  : in out Bag) is
        Of_Index  : Bag := Of_The_Bag;
        And_Index : Bag;
    begin
        To_The_Bag:= null;
        while Of_Index /= null loop
            And_Index := And_The_Bag;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            if And_Index = null then
                To_The_Bag := new Node'(The_Item  =>
Of_Index.The_Item,
                                        The_Count =>
Of_Index.The_Count,
```

```
                                            Next      => To_The_Bag);
            elsif Of_Index.The_Count > And_Index.The_Count then
                To_The_Bag := new Node'(The_Item  =>
Of_Index.The_Item,
                                            The_Count =>
Of_Index.The_Count -
And_Index.The_Count,
                                            Next      => To_The_Bag);
            end if;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Difference;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left   : in Bag;
                        Right  : in Bag;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Bag : in Bag;
                         Result  : out Natural) is
    begin
        Result := Extent_Of(The_Bag);
    end Extent_Of;

    procedure Unique_Extent_Of (The_Bag : in Bag;
                                Result  : out Natural) is
    begin
        Result := Unique_Extent_Of (The_Bag);
    end Unique_Extent_Of;

    procedure Number_Of (The_Item   : in Item;
                         In_The_Bag : in Bag;
                         Result     : out Positive) is
    begin
            Result := Number_Of(The_Item,In_The_Bag);
    end Number_Of;

    procedure Is_Empty (The_Bag : in Bag;
                        Result  : out Boolean) is
    begin
        Result := Is_Empty(The_Bag);
    end Is_Empty;

    procedure Is_A_Member (The_Item   : in Item;
                           Of_The_Bag : in Bag;
                           Result     : out Boolean) is
    begin
        Result := Is_A_Member(The_Item,Of_The_Bag);
    end Is_A_Member;

    procedure Is_A_Subset (Left   : in Bag;
                           Right  : in Bag;
                           Result : out Boolean) is
    begin
        Result := Is_A_Subset(Left,Right);
    end Is_A_Subset;

    procedure Is_A_Proper_Subset (Left   : in Bag;
                                  Right  : in Bag;
                                  Result : out Boolean) is
    begin
        Result := Is_A_Proper_Subset(Left,Right);
    end Is_A_Proper_Subset;

--  end of modification

    function Is_Equal (Left  : in Bag;
                       Right : in Bag) return Boolean is
        Left_Count  : Natural := 0;
        Right_Count : Natural := 0;
        Left_Index  : Bag     := Left;
        Right_Index : Bag;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            elsif Left_Index.The_Count /= Right_Index.The_Count then
                return False;
            else
                Left_Count := Left_Count + 1;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Right_Count := Right_Count + 1;
            Right_Index := Right_Index.Next;
        end loop;
        return (Left_Count = Right_Count);
    end Is_Equal;

    function Extent_Of (The_Bag : in Bag) return Natural is
        Count : Natural := 0;
        Index : Bag     := The_Bag;
    begin
        while Index /= null loop
            Count := Count + Index.The_Count;
            Index := Index.Next;
        end loop;
        return Count;
    end Extent_Of;

    function Unique_Extent_Of (The_Bag : in Bag) return Natural is
        Count : Natural := 0;
        Index : Bag     := The_Bag;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Unique_Extent_Of;

    function Number_Of (The_Item   : in Item;
                        In_The_Bag : in Bag) return Positive is
        Index : Bag := In_The_Bag;
    begin
        while Index /= null loop
            if The_Item = Index.The_Item then
                return Index.The_Count;
            else
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Bag;
    end Number_Of;

    function Is_Empty (The_Bag : in Bag) return Boolean is
    begin
        return (The_Bag = null);
    end Is_Empty;

    function Is_A_Member (The_Item   : in Item;
                          Of_The_Bag : in Bag) return Boolean is
        Index : Bag := Of_The_Bag;
    begin
        while Index /= null loop
            if The_Item = Index.The_Item then
                return True;
            end if;
            Index := Index.Next;
        end loop;
        return False;
    end Is_A_Member;

    function Is_A_Subset (Left  : in Bag;
                          Right : in Bag) return Boolean is
        Left_Index  : Bag := Left;
        Right_Index : Bag;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            elsif Left_Index.The_Count > Right_Index.The_Count then
                return False;
            else
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        return True;
    end Is_A_Subset;

    function Is_A_Proper_Subset (Left  : in Bag;
                                 Right : in Bag) return Boolean is
        Unique_Left_Count  : Natural := 0;
        Unique_Right_Count : Natural := 0;
        Total_Left_Count   : Natural := 0;
        Total_Right_Count  : Natural := 0;
        Left_Index         : Bag     := Left;
        Right_Index        : Bag;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            elsif Left_Index.The_Count > Right_Index.The_Count then
                return False;
            else
                Unique_Left_Count := Unique_Left_Count + 1;
                Total_Left_Count := Total_Left_Count +
Left_Index.The_Count;
                Left_Index := Left_Index.Next;
            end if;
```

```
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Unique_Right_Count := Unique_Right_Count + 1;
            Total_Right_Count := Total_Right_Count +
Right_Index.The_Count;
            Right_Index := Right_Index.Next;
        end loop;
        if Unique_Left_Count < Unique_Right_Count then
            return True;
        elsif Unique_Left_Count > Unique_Right_Count then
            return False;
        else
            return (Total_Left_Count < Total_Right_Count);
        end if;
    end Is_A_Proper_Subset;

end Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator;
```

BAG SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Bag : Bag
    OUTPUT
      From_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Bag : Bag,
      And_The_Bag : Bag,
      To_The_Bag : Bag
    OUTPUT
      To_The_Bag : Bag
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Unique_Extent_Of
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Bag : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Bag,
      Right : Bag
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_Not_In_Bag
  END

END
IMPLEMENTATION ADA
Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator
END
```

LIST OBJ3 SPECIFICATIONS

```
obj LIST[X :: TRIV] is sort List .
   protecting NAT .

*** constructors

   op create        :           -> List .
   op copy          : List List -> List .
   op clear         :      List -> List .
   op construct     : Elt  List -> List .
   op sethead       : List Elt  -> List .
***   op swaptail      : List List -> List List.
***   cannot be implemented

*** accessors

   op isequal       : List List -> Bool .
   op lengthof      :      List -> Nat .
   op isnull        :      List -> Bool .
   op headof        :      List -> Elt .
   op tailof        :      List -> List .
***   op predecessorof :      List -> List .

*** exceptions

   op overflow      : -> List .
   op listisnull    : -> List .
   op listisnull    : -> Elt .
   op notathead     : -> List .

*** variables declaration

   var L L1 : List .
   var E E1 : Elt .

*** axioms

   eq copy(L,L1) = L .
   eq clear(L) = create .
   eq sethead(create,E) = listisnull .
   eq sethead(construct(E,L),E1) = construct(E1,create) .
   eq isequal(L,L1) = L == L1 .
   eq lengthof(create) = 0 .
   eq lengthof(construct(E,L)) = 1 + lengthof(L) .
   eq isnull(L) = L == create .
   eq headof(create) = listisnull .
   eq headof(construct(E,L)) = E .
   eq tailof(create) = create .
   eq tailof(construct(E,L)) = L .
***   eq predecessorof(create) = listisnull .
***   eq predecessorof(construct(E,L)) = listisnull .

endo
```

LISTS PROFILE CODES

OPERATORS	*SIGNATURES*	*PROFILE CODES*
COPY	A B -> B	3211
CLEAR	A -> A	2201
CONSTRUCT	A B -> B	3211
SET_HEAD	A B -> A	3211
IS_EQUAL	A B -> C	330
LENGTH_OF	A -> B	220
IS_NULL	A -> B	220
HEAD_OF	A -> B	220
TAIL_OF	A -> B	220
PREDECESSOR_OF	A ->B	220

SET OF PROFILE: {3211,2201,330,220}

LIST DOUBLE BOUNDED MANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
    The_Size : in Positive;
package List_Double_Bounded_Managed is

    type List is private;

    Null_List : constant List;

    procedure Copy      (From_The_List : in     List;
                         To_The_List   : in out List);
    procedure Clear     (The_List      : in out List);
    procedure Construct (The_Item      : in     Item;
                         And_The_List  : in out List);
    procedure Set_Head  (Of_The_List   : in out List;
                         To_The_Item   : in     Item);
    procedure Swap_Tail (Of_The_List   : in out List;
                         And_The_List  : in out List);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal       (Left     : in List;
                              Right    : in List;
                              Result   : out Boolean);
    procedure Length_Of      (The_List : in List;
                              Result   : out Natural);
    procedure Is_Null        (The_List : in List;
                              Result   : out Boolean);
    procedure Head_Of        (The_List : in List;
                              Result   : out Item);
    procedure Tail_Of        (The_List : in List;
                              Result   : out List);
    procedure Predecessor_Of (The_List : in List;
                              Result   : out List);

--  end of modification

    function Is_Equal       (Left     : in List;
                             Right    : in List) return Boolean;
    function Length_Of      (The_List : in List) return Natural;
    function Is_Null        (The_List : in List) return Boolean;
    function Head_Of        (The_List : in List) return Item;
    function Tail_Of        (The_List : in List) return List;
    function Predecessor_Of (The_List : in List) return List;

    Overflow     : exception;
    List_Is_Null : exception;
    Not_At_Head  : exception;

private
    type List is
        record
            The_Head : Natural := 0;
        end record;
    Null_List : constant List := List'(The_Head => 0);
end List_Double_Bounded_Managed;
```

LIST DOUBLE BOUNDED MANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body List_Double_Bounded_Managed is

    type Node is
        record
            Previous : List;
            The_Item : Item;
            Next     : List;
        end record;

    Heap : array(Positive range 1 .. The_Size) of Node;

    Free_List : List;

    procedure Free (The_List : in out List) is
        Temporary_Node : List;
    begin
        while The_List /= Null_List loop
            Temporary_Node := The_List;
            The_List := Heap(The_List.The_Head).Next;
            Heap(Temporary_Node.The_Head).Previous := Null_List;
            Heap(Temporary_Node.The_Head).Next := Free_List;
            Free_List := Temporary_Node;
        end loop;
    end Free;

    function New_Item return List is
        Temporary_Node : List;
    begin
        if Free_List = Null_List then
            raise Storage_Error;
        else
            Temporary_Node := Free_List;
            Free_List := Heap(Free_List.The_Head).Next;
            Heap(Temporary_Node.The_Head).Next := Null_List;
            return Temporary_Node;
        end if;
    end New_Item;

    procedure Copy (From_The_List : in     List;
                    To_The_List   : in out List) is
        From_Index : List := From_The_List;
        To_Index   : List;
    begin
        Free(To_The_List);
        if From_The_List /= Null_List then
            To_The_List := New_Item;
            Heap(To_The_List.The_Head).The_Item :=
              Heap(From_Index.The_Head).The_Item;
            To_Index := To_The_List;
            From_Index := Heap(From_Index.The_Head).Next;
            while From_Index /= Null_List loop
                Heap(To_Index.The_Head).Next := New_Item;
                Heap(Heap(To_Index.The_Head).Next.The_Head).Previous
:=
                  To_Index;
                To_Index := Heap(To_Index.The_Head).Next;
                Heap(To_Index.The_Head).The_Item :=
                  Heap(From_Index.The_Head).The_Item;
                From_Index := Heap(From_Index.The_Head).Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_List : in out List) is
    begin
        Free(The_List);
    end Clear;

    procedure Construct (The_Item     : in     Item;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        if And_The_List = Null_List then
            And_The_List := New_Item;
            Heap(And_The_List.The_Head).The_Item := The_Item;
        elsif Heap(And_The_List.The_Head).Previous = Null_List then
            Temporary_Node := New_Item;
            Heap(Temporary_Node.The_Head).The_Item := The_Item;
            Heap(Temporary_Node.The_Head).Next := And_The_List;
            Heap(And_The_List.The_Head).Previous := Temporary_Node;
            And_The_List := Temporary_Node;
        else
            raise Not_At_Head;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Construct;

    procedure Set_Head (Of_The_List : in out List;
                        To_The_Item : in     Item) is
    begin
        Heap(Of_The_List.The_Head).The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Set_Head;

    procedure Swap_Tail (Of_The_List  : in out List;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        if And_The_List = Null_List then
            if Heap(Of_The_List.The_Head).Next /= Null_List then
                Temporary_Node := Heap(Of_The_List.The_Head).Next;
                Heap(Temporary_Node.The_Head).Previous := Null_List;
                Heap(Of_The_List.The_Head).Next := Null_List;
                And_The_List := Temporary_Node;
            end if;
        elsif Heap(And_The_List.The_Head).Previous = Null_List then
            if Heap(Of_The_List.The_Head).Next /= Null_List then
                Temporary_Node := Heap(Of_The_List.The_Head).Next;
                Heap(Temporary_Node.The_Head).Previous := Null_List;
                Heap(Of_The_List.The_Head).Next := And_The_List;
                Heap(And_The_List.The_Head).Previous := Of_The_List;
                And_The_List := Temporary_Node;
            else
                Heap(And_The_List.The_Head).Previous := Of_The_List;
                Heap(Of_The_List.The_Head).Next := And_The_List;
                And_The_List := Null_List;
            end if;
        else
            raise Not_At_Head;
        end if;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Swap_Tail;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal       (Left     : in List;
                              Right    : in List;
                              Result   : out Boolean) is
    begin
        Result :=  Is_Equal   (Left,Right);
    end Is_Equal;

    procedure Length_Of      (The_List : in List;
                              Result   : out Natural) is
    begin
        Result := Length_Of      (The_List);
    end Length_Of;

    procedure Is_Null        (The_List : in List;
                              Result   : out Boolean) is
    begin
        Result := Is_Null        (The_List);
    end Is_Null;

    procedure Head_Of        (The_List : in List;
                              Result   : out Item) is
    begin
        Result := Head_Of        (The_List);
    end Head_Of;

    procedure Tail_Of        (The_List : in List;
                              Result   : out List) is
    begin
        Result := Tail_Of        (The_List);
    end Tail_Of;

    procedure Predecessor_Of (The_List : in List;
                              Result   : out List) is
    begin
        Result := Predecessor_Of (The_List);
    end Predecessor_Of;

--  end of modification

    function Is_Equal (Left  : in List;
                       Right : in List) return Boolean is
        Left_Index  : List := Left;
        Right_Index : List := Right;
    begin
        while Left_Index /= Null_List loop
            if Heap(Left_Index.The_Head).The_Item /=
```

```
            Heap(Right_Index.The_Head).The_Item then
                return False;
            end if;
            Left_Index := Heap(Left_Index.The_Head).Next;
            Right_Index := Heap(Right_Index.The_Head).Next;
        end loop;
        return (Right_Index = Null_List);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_List : in List) return Natural is
        Count : Natural := 0;
        Index : List    := The_List;
    begin
        while Index /= Null_List loop
            Count := Count + 1;
            Index := Heap(Index.The_Head).Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Null (The_List : in List) return Boolean is
    begin
        return (The_List = Null_List);
    end Is_Null;

    function Head_Of (The_List : in List) return Item is
    begin
        return Heap(The_List.The_Head).The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Head_Of;

    function Tail_Of (The_List : in List) return List is
    begin
        return Heap(The_List.The_Head).Next;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Tail_Of;

    function Predecessor_Of (The_List : in List) return List is
    begin
        return Heap(The_List.The_Head).Previous;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Predecessor_Of;

begin
    Free_List.The_Head := 1;
    for Index in 1 .. (The_Size - 1) loop
        Heap(Index).Previous := Null_List;
        Heap(Index).Next := List'(The_Head => (Index + 1));
    end loop;
    Heap(The_Size).Next := Null_List;
end List_Double_Bounded_Managed;
```

LIST DOUBLE BOUNDED MANAGED

PSDL

```
TYPE List_Double_Bounded_Managed
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_List : List,
      To_The_List : List
    OUTPUT
      To_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_List : List
    OUTPUT
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Set_Head
  SPECIFICATION
    INPUT
      Of_The_List : List,
      To_The_Item : Item
    OUTPUT
      Of_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Swap_Tail
  SPECIFICATION
    INPUT
      Of_The_List : List,
      And_The_List : List
    OUTPUT
      Of_The_List : List,
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : List,
      Right : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Head_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Tail_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Predecessor_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

END
IMPLEMENTATION ADA List_Double_Bounded_Managed
END
```

LIST DOUBLE UNBOUNDED MANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
package List_Double_Unbounded_Managed is

    type List is private;

    Null_List : constant List;

    procedure Copy      (From_The_List : in     List;
                         To_The_List   : in out List);
    procedure Clear     (The_List      : in out List);
    procedure Construct (The_Item      : in     Item;
                         And_The_List  : in out List);
    procedure Set_Head  (Of_The_List   : in out List;
                         To_The_Item   : in     Item);
    procedure Swap_Tail (Of_The_List   : in out List;
                         And_The_List  : in out List);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal        (Left     : in List;
                               Right    : in List;
                               Result   : out Boolean);
    procedure Length_Of       (The_List : in List;
                               Result   : out Natural);
    procedure Is_Null         (The_List : in List;
                               Result   : out Boolean);
    procedure Head_Of         (The_List : in List;
                               Result   : out Item);
    procedure Tail_Of         (The_List : in List;
                               Result   : out List);
    procedure Predecessor_Of (The_List : in List;
                               Result   : out List);

--  end of modification

    function Is_Equal       (Left     : in List;
                             Right    : in List) return Boolean;
    function Length_Of      (The_List : in List) return Natural;
    function Is_Null        (The_List : in List) return Boolean;
    function Head_Of        (The_List : in List) return Item;
    function Tail_Of        (The_List : in List) return List;
    function Predecessor_Of (The_List : in List) return List;

    Overflow     : exception;
    List_Is_Null : exception;
    Not_At_Head  : exception;

private
    type Node;
    type List is access Node;
    Null_List : constant List := null;
end List_Double_Unbounded_Managed;
```

LIST DOUBLE UNBOUNDED MANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body List_Double_Unbounded_Managed is

    type Node is
        record
            Previous : List;
            The_Item : Item;
            Next     : List;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        The_Node.Previous := null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     List) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return List is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                  (Item        => Node,
                                   Pointer     => List,
                    Free        => Free,
                    Set_Pointer => Set_Next,
                    Pointer_Of  => Next_Of);

    procedure Copy (From_The_List : in     List;
                    To_The_List   : in out List) is
        From_Index : List := From_The_List;
        To_Index   : List;
    begin
        Node_Manager.Free(To_The_List);
        if From_The_List /= null then
            To_The_List := Node_Manager.New_Item;
            To_The_List.The_Item := From_Index.The_Item;
            To_Index := To_The_List;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.Previous := To_Index;
                To_Index := To_Index.Next;
                To_Index.The_Item := From_Index.The_Item;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_List : in out List) is
    begin
        Node_Manager.Free(The_List);
    end Clear;

    procedure Construct (The_Item     : in     Item;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        if And_The_List = null then
            And_The_List := Node_Manager.New_Item;
            And_The_List.The_Item := The_Item;
        elsif And_The_List.Previous = null then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := The_Item;
            Temporary_Node.Next := And_The_List;
            And_The_List.Previous := Temporary_Node;
            And_The_List := Temporary_Node;
        else
            raise Not_At_Head;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Construct;

    procedure Set_Head (Of_The_List : in out List;
                        To_The_Item : in     Item) is
    begin
        Of_The_List.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Set_Head;

    procedure Swap_Tail (Of_The_List  : in out List;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        if And_The_List = null then
            if Of_The_List.Next /= null then
                Temporary_Node := Of_The_List.Next;
                Temporary_Node.Previous := null;
                Of_The_List.Next := null;
                And_The_List := Temporary_Node;
            end if;
        elsif And_The_List.Previous = null then
            if Of_The_List.Next /= null then
                Temporary_Node := Of_The_List.Next;
                Of_The_List.Next.Previous := null;
                Of_The_List.Next := And_The_List;
                And_The_List.Previous := Of_The_List;
                And_The_List := Temporary_Node;
            else
                And_The_List.Previous := Of_The_List;
                Of_The_List.Next := And_The_List;
                And_The_List := null;
            end if;
        else
            raise Not_At_Head;
        end if;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Swap_Tail;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal       (Left     : in List;
                              Right    : in List;
                              Result   : out Boolean) is
    begin
        Result :=  Is_Equal   (Left,Right);
    end Is_Equal;

    procedure Length_Of      (The_List : in List;
                              Result   : out Natural) is
    begin
        Result := Length_Of      (The_List);
    end Length_Of;

    procedure Is_Null        (The_List : in List;
                              Result   : out Boolean) is
    begin
        Result := Is_Null        (The_List);
    end Is_Null;

    procedure Head_Of        (The_List : in List;
                              Result   : out Item) is
    begin
        Result := Head_Of        (The_List);
    end Head_Of;

    procedure Tail_Of        (The_List : in List;
                              Result   : out List) is
    begin
        Result := Tail_Of        (The_List);
    end Tail_Of;

    procedure Predecessor_Of (The_List : in List;
                              Result   : out List) is
    begin
        Result := Predecessor_Of (The_List);
    end Predecessor_Of;

--  end of modification

    function Is_Equal (Left  : in List;
                       Right : in List) return Boolean is
        Left_Index  : List := Left;
        Right_Index : List := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
```

```
end Is_Equal;

function Length_Of (The_List : in List) return Natural is
    Count : Natural := 0;
    Index : List    := The_List;
begin
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Length_Of;

function Is_Null (The_List : in List) return Boolean is
begin
    return (The_List = null);
end Is_Null;

function Head_Of (The_List : in List) return Item is
begin
    return The_List.The_Item;
exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Head_Of;

    function Tail_Of (The_List : in List) return List is
    begin
        return The_List.Next;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Tail_Of;

    function Predecessor_Of (The_List : in List) return List is
    begin
        return The_List.Previous;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Predecessor_Of;

end List_Double_Unbounded_Managed;
```

LIST DOUBLE UNBOUNDED MANAGED

PSDL

```
TYPE List_Double_Unbounded_Managed
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_List : List,
      To_The_List : List
    OUTPUT
      To_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_List : List
    OUTPUT
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Set_Head
  SPECIFICATION
    INPUT
      Of_The_List : List,
      To_The_Item : Item
    OUTPUT
      Of_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Swap_Tail
  SPECIFICATION
    INPUT
      Of_The_List : List,
      And_The_List : List
    OUTPUT
      Of_The_List : List,
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : List,
      Right : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Head_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Tail_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Predecessor_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

END
IMPLEMENTATION ADA List_Double_Unbounded_Managed
END
```

LIST DOUBLE UNBOUNDED UNMANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
package List_Double_Unbounded_Unmanaged is

    type List is private;

    Null_List : constant List;

    procedure Copy      (From_The_List : in     List;
                         To_The_List   : in out List);
    procedure Clear     (The_List      : in out List);
    procedure Construct (The_Item      : in     Item;
                         And_The_List  : in out List);
    procedure Set_Head  (Of_The_List   : in out List;
                         To_The_Item   : in     Item);
    procedure Swap_Tail (Of_The_List   : in out List;
                         And_The_List  : in out List);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal         (Left     : in List;
                                Right    : in List;
                                Result   : out Boolean);
    procedure Length_Of        (The_List : in List;
                                Result   : out Natural);
    procedure Is_Null          (The_List : in List;
                                Result   : out Boolean);
    procedure Head_Of          (The_List : in List;
                                Result   : out Item);
    procedure Tail_Of          (The_List : in List;
                                Result   : out List);
    procedure Predecessor_Of (The_List : in List;
                                Result   : out List);

--  end of modification

    function Is_Equal       (Left     : in List;
                             Right    : in List) return Boolean;
    function Length_Of      (The_List : in List) return Natural;
    function Is_Null        (The_List : in List) return Boolean;
    function Head_Of        (The_List : in List) return Item;
    function Tail_Of        (The_List : in List) return List;
    function Predecessor_Of (The_List : in List) return List;

    Overflow     : exception;
    List_Is_Null : exception;
    Not_At_Head  : exception;

private
    type Node;
    type List is access Node;
    Null_List : constant List := null;
end List_Double_Unbounded_Unmanaged;
```

LIST DOUBLE UNBOUNDED UNMANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body List_Double_Unbounded_Unmanaged is

    type Node is
        record
            Previous : List;
            The_Item : Item;
            Next     : List;
        end record;

    procedure Copy (From_The_List : in     List;
                    To_The_List   : in out List) is
        From_Index : List := From_The_List;
        To_Index   : List;
    begin
        if From_The_List = null then
            To_The_List := null;
        else
            To_The_List := new Node'(Previous => null,
                                     The_Item => From_Index.The_Item,
                                     Next     => null);
            To_Index := To_The_List;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(Previous => To_Index,
                                           The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_List : in out List) is
    begin
        The_List := null;
    end Clear;

    procedure Construct (The_Item     : in     Item;
                         And_The_List : in out List) is
    begin
        if And_The_List = null then
            And_The_List := new Node'(Previous => null,
                                      The_Item => The_Item,
                                      Next     => null);
        elsif And_The_List.Previous = null then
            And_The_List := new Node'(Previous => null,
                                      The_Item => The_Item,
                                      Next     => And_The_List);
            And_The_List.Next.Previous := And_The_List;
        else
            raise Not_At_Head;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Construct;

    procedure Set_Head (Of_The_List : in out List;
                        To_The_Item : in     Item) is
    begin
        Of_The_List.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Set_Head;

    procedure Swap_Tail (Of_The_List  : in out List;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        if And_The_List = null then
            if Of_The_List.Next /= null then
                Temporary_Node := Of_The_List.Next;
                Temporary_Node.Previous := null;
                Of_The_List.Next := null;
                And_The_List := Temporary_Node;
            end if;
        elsif And_The_List.Previous = null then
            if Of_The_List.Next /= null then
                Temporary_Node := Of_The_List.Next;
                Temporary_Node.Previous := null;
                Of_The_List.Next := And_The_List;
                And_The_List.Previous := Of_The_List;
                And_The_List := Temporary_Node;
            else
                And_The_List.Previous := Of_The_List;
                Of_The_List.Next := And_The_List;
                And_The_List := null;
            end if;
        else
            raise Not_At_Head;
        end if;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Swap_Tail;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal        (Left     : in List;
                               Right    : in List;
                               Result   : out Boolean) is
    begin
        Result :=  Is_Equal   (Left,Right);
    end Is_Equal;

    procedure Length_Of       (The_List : in List;
                               Result   : out Natural) is
    begin
        Result := Length_Of       (The_List);
    end Length_Of;

    procedure Is_Null         (The_List : in List;
                               Result   : out Boolean) is
    begin
        Result := Is_Null         (The_List);
    end Is_Null;

    procedure Head_Of         (The_List : in List;
                               Result   : out Item) is
    begin
        Result := Head_Of         (The_List);
    end Head_Of;

    procedure Tail_Of         (The_List : in List;
                               Result   : out List) is
    begin
        Result := Tail_Of         (The_List);
    end Tail_Of;

    procedure Predecessor_Of (The_List : in List;
                              Result   : out List) is
    begin
        Result := Predecessor_Of (The_List);
    end Predecessor_Of;

--  end of modification

    function Is_Equal (Left  : in List;
                       Right : in List) return Boolean is
        Left_Index  : List := Left;
        Right_Index : List := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_List : in List) return Natural is
        Count : Natural := 0;
        Index : List    := The_List;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Null (The_List : in List) return Boolean is
    begin
        return (The_List = null);
    end Is_Null;

    function Head_Of (The_List : in List) return Item is
    begin
        return The_List.The_Item;
    exception
```

```
    when Constraint_Error =>
        raise List_Is_Null;
end Head_Of;

function Tail_Of (The_List : in List) return List is
begin
    return The_List.Next;
exception
    when Constraint_Error =>
        raise List_Is_Null;
end Tail_Of;

    function Predecessor_Of (The_List : in List) return List is
    begin
        return The_List.Previous;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Predecessor_Of;

end List_Double_Unbounded_Unmanaged;
```

LIST DOUBLE UNBOUNDED UNMANAGED

PSDL

```
TYPE List_Double_Unbounded_Unmanaged
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_List : List,
      To_The_List : List
    OUTPUT
      To_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_List : List
    OUTPUT
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Set_Head
  SPECIFICATION
    INPUT
      Of_The_List : List,
      To_The_Item : Item
    OUTPUT
      Of_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Swap_Tail
  SPECIFICATION
    INPUT
      Of_The_List : List,
      And_The_List : List
    OUTPUT
      Of_The_List : List,
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : List,
      Right : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Head_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Tail_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

  OPERATOR Predecessor_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null, Not_At_Head
  END

END
IMPLEMENTATION ADA List_Double_Unbounded_Unmanaged
END
```

LIST SINGLE BOUNDED MANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
    The_Size : in Positive;
package List_Single_Bounded_Managed is

    type List is private;

    Null_List : constant List;

    procedure Copy      (From_The_List : in      List;
                         To_The_List   : in out List);
    procedure Clear     (The_List      : in out List);
    procedure Construct (The_Item      : in      Item;
                         And_The_List  : in out List);
    procedure Set_Head  (Of_The_List   : in out List;
                         To_The_Item   : in      Item);
    procedure Swap_Tail (Of_The_List   : in out List;
                         And_The_List  : in out List);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal         (Left     : in List;
                                Right    : in List;
                                Result   : out Boolean);
    procedure Length_Of        (The_List : in List;
                                Result   : out Natural);
    procedure Is_Null          (The_List : in List;
                                Result   : out Boolean);
    procedure Head_Of          (The_List : in List;
                                Result   : out Item);
    procedure Tail_Of          (The_List : in List;
                                Result   : out List);

--  end of modification

    function Is_Equal  (Left     : in List;
                        Right    : in List) return Boolean;
    function Length_Of (The_List : in List) return Natural;
    function Is_Null   (The_List : in List) return Boolean;
    function Head_Of   (The_List : in List) return Item;
    function Tail_Of   (The_List : in List) return List;

    Overflow     : exception;
    List_Is_Null : exception;

private
    type List is
        record
            The_Head : Natural := 0;
        end record;
    Null_List : constant List := List'(The_Head => 0);
end List_Single_Bounded_Managed;
```

LIST SINGLE BOUNDED MANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body List_Single_Bounded_Managed is

    type Node is
        record
            The_Item : Item;
            Next     : List;
        end record;

    Heap : array(Positive range 1 .. The_Size) of Node;

    Free_List : List;

    procedure Free (The_List : in out List) is
        Temporary_Node : List;
    begin
        while The_List /= Null_List loop
            Temporary_Node := The_List;
            The_List := Heap(The_List.The_Head).Next;
            Heap(Temporary_Node.The_Head).Next := Free_List;
            Free_List := Temporary_Node;
        end loop;
    end Free;

    function New_Item return List is
        Temporary_Node : List;
    begin
        if Free_List = Null_List then
            raise Storage_Error;
        else
            Temporary_Node := Free_List;
            Free_List := Heap(Free_List.The_Head).Next;
            Heap(Temporary_Node.The_Head).Next := Null_List;
            return Temporary_Node;
        end if;
    end New_Item;

    procedure Copy (From_The_List : in     List;
                    To_The_List   : in out List) is
        From_Index : List := From_The_List;
        To_Index   : List;
    begin
        Free(To_The_List);
        if From_The_List /= Null_List then
            To_The_List := New_Item;
            Heap(To_The_List.The_Head).The_Item :=
              Heap(From_Index.The_Head).The_Item;
            To_Index := To_The_List;
            From_Index := Heap(From_Index.The_Head).Next;
            while From_Index /= Null_List loop
                Heap(To_Index.The_Head).Next := New_Item;
                To_Index := Heap(To_Index.The_Head).Next;
                Heap(To_Index.The_Head).The_Item :=
                  Heap(From_Index.The_Head).The_Item;
                From_Index := Heap(From_Index.The_Head).Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_List : in out List) is
    begin
        Free(The_List);
    end Clear;

    procedure Construct (The_Item     : in     Item;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        Temporary_Node := New_Item;
        Heap(Temporary_Node.The_Head).The_Item := The_Item;
        Heap(Temporary_Node.The_Head).Next := And_The_List;
        And_The_List := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Construct;

    procedure Set_Head (Of_The_List : in out List;
                        To_The_Item : in     Item) is
    begin
        Heap(Of_The_List.The_Head).The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Set_Head;

    procedure Swap_Tail (Of_The_List  : in out List;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        Temporary_Node := Heap(Of_The_List.The_Head).Next;
        Heap(Of_The_List.The_Head).Next := And_The_List;
        And_The_List := Temporary_Node;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Swap_Tail;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal      (Left     : in List;
                             Right    : in List;
                             Result   : out Boolean) is
    begin
        Result :=  Is_Equal   (Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_List : in List;
                             Result   : out Natural) is
    begin
        Result := Length_Of      (The_List);
    end Length_Of;

    procedure Is_Null       (The_List : in List;
                             Result   : out Boolean) is
    begin
        Result := Is_Null        (The_List);
    end Is_Null;

    procedure Head_Of       (The_List : in List;
                             Result   : out Item) is
    begin
        Result := Head_Of        (The_List);
    end Head_Of;

    procedure Tail_Of       (The_List : in List;
                             Result   : out List) is
    begin
        Result := Tail_Of        (The_List);
    end Tail_Of;

--  end of modification

    function Is_Equal (Left  : in List;
                       Right : in List) return Boolean is
        Left_Index  : List := Left;
        Right_Index : List := Right;
    begin
        while Left_Index /= Null_List loop
            if Heap(Left_Index.The_Head).The_Item /=
              Heap(Right_Index.The_Head).The_Item then
                return False;
            end if;
            Left_Index := Heap(Left_Index.The_Head).Next;
            Right_Index := Heap(Right_Index.The_Head).Next;
        end loop;
        return (Right_Index = Null_List);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_List : in List) return Natural is
        Count : Natural := 0;
        Index : List    := The_List;
    begin
        while Index /= Null_List loop
            Count := Count + 1;
            Index := Heap(Index.The_Head).Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Null (The_List : in List) return Boolean is
    begin
        return (The_List = Null_List);
    end Is_Null;

    function Head_Of (The_List : in List) return Item is
    begin
        return Heap(The_List.The_Head).The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Head_Of;

    function Tail_Of (The_List : in List) return List is
    begin
```

```
        return Heap(The_List.The_Head).Next;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Tail_Of;

begin
```

```
        Free_List.The_Head := 1;
        for Index in 1 .. (The_Size - 1) loop
            Heap(Index).Next := List'(The_Head => (Index + 1));
        end loop;
        Heap(The_Size).Next := Null_List;
    end List_Single_Bounded_Managed;
```

LIST SINGLE BOUNDED MANAGED

PSDL

```
TYPE List_Single_Bounded_Managed
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_List : List,
      To_The_List : List
    OUTPUT
      To_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_List : List
    OUTPUT
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Set_Head
  SPECIFICATION
    INPUT
      Of_The_List : List,
      To_The_Item : Item
    OUTPUT
      Of_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Swap_Tail
  SPECIFICATION
    INPUT
      Of_The_List : List,
      And_The_List : List
    OUTPUT
      Of_The_List : List,
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : List,
      Right : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Head_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Tail_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

END
IMPLEMENTATION ADA List_Single_Bounded_Managed
END
```

LIST SINGLE UNBOUNDED MANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
package List_Single_Unbounded_Managed is

    type List is private;

    Null_List : constant List;

    procedure Copy      (From_The_List : in     List;
                         To_The_List   : in out List);
    procedure Clear     (The_List      : in out List);
    procedure Construct (The_Item      : in     Item;
                         And_The_List  : in out List);
    procedure Set_Head  (Of_The_List   : in out List;
                         To_The_Item   : in     Item);
    procedure Swap_Tail (Of_The_List   : in out List;
                         And_The_List  : in out List);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal       (Left     : in List;
                              Right    : in List;
                              Result   : out Boolean);
    procedure Length_Of      (The_List : in List;
                              Result   : out Natural);
    procedure Is_Null        (The_List : in List;
                              Result   : out Boolean);
    procedure Head_Of        (The_List : in List;
                              Result   : out Item);
    procedure Tail_Of        (The_List : in List;
                              Result   : out List);

--  end of modification

    function Is_Equal  (Left     : in List;
                        Right    : in List) return Boolean;
    function Length_Of (The_List : in List) return Natural;
    function Is_Null   (The_List : in List) return Boolean;
    function Head_Of   (The_List : in List) return Item;
    function Tail_Of   (The_List : in List) return List;

    Overflow     : exception;
    List_Is_Null : exception;

private
    type Node;
    type List is access Node;
    Null_List : constant List := null;
end List_Single_Unbounded_Managed;
```

LIST SINGLE UNBOUNDED MANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body List_Single_Unbounded_Managed is

    type Node is
        record
            The_Item : Item;
            Next     : List;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     List) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return List is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => List,
                    Free        => Free,
                    Set_Pointer => Set_Next,
                    Pointer_Of  => Next_Of);

    procedure Copy (From_The_List : in     List;
                    To_The_List   : in out List) is
        From_Index : List := From_The_List;
        To_Index   : List;
    begin
        Node_Manager.Free(To_The_List);
        if From_The_List /= null then
            To_The_List := Node_Manager.New_Item;
            To_The_List.The_Item := From_Index.The_Item;
            To_Index := To_The_List;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index := To_Index.Next;
                To_Index.The_Item := From_Index.The_Item;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_List : in out List) is
    begin
        Node_Manager.Free(The_List);
    end Clear;

    procedure Construct (The_Item     : in     Item;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.Next := And_The_List;
        And_The_List := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Construct;

    procedure Set_Head (Of_The_List : in out List;
                        To_The_Item : in     Item) is
    begin
        Of_The_List.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Set_Head;

    procedure Swap_Tail (Of_The_List  : in out List;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        Temporary_Node := Of_The_List.Next;
        Of_The_List.Next := And_The_List;
        And_The_List := Temporary_Node;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Swap_Tail;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal         (Left     : in List;
                                Right    : in List;
                                Result   : out Boolean) is
    begin
        Result :=  Is_Equal   (Left,Right);
    end Is_Equal;

    procedure Length_Of        (The_List : in List;
                                Result   : out Natural) is
    begin
        Result := Length_Of       (The_List);
    end Length_Of;

    procedure Is_Null          (The_List : in List;
                                Result   : out Boolean) is
    begin
        Result := Is_Null         (The_List);
    end Is_Null;

    procedure Head_Of          (The_List : in List;
                                Result   : out Item) is
    begin
        Result := Head_Of         (The_List);
    end Head_Of;

    procedure Tail_Of          (The_List : in List;
                                Result   : out List) is
    begin
        Result := Tail_Of         (The_List);
    end Tail_Of;

--  end of modification

    function Is_Equal (Left  : in List;
                       Right : in List) return Boolean is
        Left_Index  : List := Left;
        Right_Index : List := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_List : in List) return Natural is
        Count : Natural := 0;
        Index : List    := The_List;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Null (The_List : in List) return Boolean is
    begin
        return (The_List = null);
    end Is_Null;

    function Head_Of (The_List : in List) return Item is
    begin
        return The_List.The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Head_Of;

    function Tail_Of (The_List : in List) return List is
    begin
        return The_List.Next;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Tail_Of;

end List_Single_Unbounded_Managed;
```

LIST SINGLE UNBOUNDED MANAGED

PSDL

```
TYPE List_Single_Unbounded_Managed
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_List : List,
      To_The_List : List
    OUTPUT
      To_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_List : List
    OUTPUT
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Set_Head
  SPECIFICATION
    INPUT
      Of_The_List : List,
      To_The_Item : Item
    OUTPUT
      Of_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Swap_Tail
  SPECIFICATION
    INPUT
      Of_The_List : List,
      And_The_List : List
    OUTPUT
      Of_The_List : List,
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : List,
      Right : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Head_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Tail_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

END
IMPLEMENTATION ADA List_Single_Unbounded_Managed
END
```

LIST SINGLE UNBOUNDED UNMANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
package List_Single_Unbounded_Unmanaged is

    type List is private;

    Null_List : constant List;

    procedure Copy      (From_The_List : in     List;
                         To_The_List   : in out List);
    procedure Clear     (The_List      : in out List);
    procedure Construct (The_Item      : in     Item;
                         And_The_List  : in out List);
    procedure Set_Head  (Of_The_List   : in out List;
                         To_The_Item   : in     Item);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal        (Left     : in List;
                               Right    : in List;
                               Result   : out Boolean);
    procedure Length_Of       (The_List : in List;
                               Result   : out Natural);
    procedure Is_Null         (The_List : in List;
                               Result   : out Boolean);
    procedure Head_Of         (The_List : in List;
                               Result   : out Item);
    procedure Tail_Of         (The_List : in List;
                               Result   : out List);

--  end of modification

    function Is_Equal  (Left     : in List;
                        Right    : in List) return Boolean;
    function Length_Of (The_List : in List) return Natural;
    function Is_Null   (The_List : in List) return Boolean;
    function Head_Of   (The_List : in List) return Item;
    function Tail_Of   (The_List : in List) return List;

    Overflow     : exception;
    List_Is_Null : exception;

private
    type Node;
    type List is access Node;
    Null_List : constant List := null;
end List_Single_Unbounded_Unmanaged;
```

LIST SINGLE UNBOUNDED UNMANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body List_Single_Unbounded_Unmanaged is

    type Node is
        record
            The_Item : Item;
            Next     : List;
        end record;

    procedure Copy (From_The_List : in     List;
                    To_The_List   : in out List) is
        From_Index : List := From_The_List;
        To_Index   : List;
    begin
        if From_The_List = null then
            To_The_List := null;
        else
            To_The_List := new Node'(The_Item => From_Index.The_Item,
                                     Next     => null);
            To_Index := To_The_List;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_List : in out List) is
    begin
        The_List := null;
    end Clear;

    procedure Construct (The_Item     : in     Item;
                         And_The_List : in out List) is
    begin
        And_The_List := new Node'(The_Item => The_Item,
                                  Next     => And_The_List);
    exception
        when Storage_Error =>
            raise Overflow;
    end Construct;

    procedure Set_Head (Of_The_List : in out List;
                        To_The_Item : in     Item) is
    begin
        Of_The_List.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Set_Head;

    procedure Swap_Tail (Of_The_List  : in out List;
                         And_The_List : in out List) is
        Temporary_Node : List;
    begin
        Temporary_Node := Of_The_List.Next;
        Of_The_List.Next := And_The_List;
        And_The_List := Temporary_Node;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Swap_Tail;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal        (Left     : in List;
                               Right    : in List;
                               Result   : out Boolean) is
    begin
        Result :=  Is_Equal   (Left,Right);
    end Is_Equal;

    procedure Length_Of       (The_List : in List;
                               Result   : out Natural) is
    begin
        Result := Length_Of       (The_List);
    end Length_Of;

    procedure Is_Null         (The_List : in List;
                               Result   : out Boolean) is
    begin
        Result := Is_Null         (The_List);
    end Is_Null;

    procedure Head_Of         (The_List : in List;
                               Result   : out Item) is
    begin
        Result := Head_Of         (The_List);
    end Head_Of;

    procedure Tail_Of         (The_List : in List;
                               Result   : out List) is
    begin
        Result := Tail_Of         (The_List);
    end Tail_Of;

--  end of modification

    function Is_Equal (Left  : in List;
                       Right : in List) return Boolean is
        Left_Index  : List := Left;
        Right_Index : List := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_List : in List) return Natural is
        Count : Natural := 0;
        Index : List    := The_List;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Null (The_List : in List) return Boolean is
    begin
        return (The_List = null);
    end Is_Null;

    function Head_Of (The_List : in List) return Item is
    begin
        return The_List.The_Item;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Head_Of;

    function Tail_Of (The_List : in List) return List is
    begin
        return The_List.Next;
    exception
        when Constraint_Error =>
            raise List_Is_Null;
    end Tail_Of;

end List_Single_Unbounded_Unmanaged;
```

LIST SINGLE UNBOUNDED UNMANAGED

PSDL

```
TYPE List_Single_Unbounded_Unmanaged
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_List : List,
      To_The_List : List
    OUTPUT
      To_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_List : List
    OUTPUT
      And_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Set_Head
  SPECIFICATION
    INPUT
      Of_The_List : List,
      To_The_Item : Item
    OUTPUT
      Of_The_List : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : List,
      Right : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Head_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, List_Is_Null
  END

  OPERATOR Tail_Of
  SPECIFICATION
    INPUT
      The_List : List
    OUTPUT
      Result : List
    EXCEPTIONS
      Overflow, List_Is_Null
  END

END
IMPLEMENTATION ADA List_Single_Unbounded_Unmanaged
END
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Domain is private;
    type Ranges is private;
    with function Hash_Of (The_Domain : in Domain) return Positive;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    with procedure Hash_Of (The_Domain : in Domain;
                            Result     : out Positive);
--  end of modication

package Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator is

    type Map(The_Size : Positive) is limited private;

    procedure Copy   (From_The_Map  : in     Map;
                      To_The_Map    : in out Map);
    procedure Clear  (The_Map       : in out Map);
    procedure Bind   (The_Domain    : in     Domain;
                      And_The_Range : in     Ranges;
                      In_The_Map    : in out Map);
    procedure Unbind (The_Domain    : in     Domain;
                      In_The_Map    : in out Map);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal   (Left   : in Map;
                          Right  : in Map;
                          Result : out Boolean);
    procedure Extent_Of  (The_Map  : in Map;
                          Result   : out Natural);
    procedure Is_Empty   (The_Map  : in Map;
                                  Result    : out Boolean);
    procedure Is_Bound   (The_Domain : in Domain;
                          In_The_Map : in Map;
                          Result     : out Boolean);
    procedure Range_Of   (The_Domain : in Domain;
                          In_The_Map : in Map;
                          Result     : out Ranges);
--  end of modication

    function Is_Equal  (Left       : in Map;
                        Right      : in Map) return Boolean;
    function Extent_Of (The_Map    : in Map) return Natural;
    function Is_Empty  (The_Map    : in Map) return Boolean;
    function Is_Bound  (The_Domain : in Domain;
                        In_The_Map : in Map) return Boolean;
    function Range_Of  (The_Domain : in Domain;
                        In_The_Map : in Map) return Ranges;

    generic
        with procedure Process (The_Domain : in  Domain;
                                The_Range  : in  Ranges;
                                Continue   : out Boolean);
    procedure Iterate (Over_The_Map : in Map);

    Overflow            : exception;
    Domain_Is_Not_Bound : exception;
    Multiple_Binding    : exception;

private
    type State is (Empty, Deleted, Bound);
    type Node is
        record
            The_State  : State := Empty;
            The_Domain : Domain;
            The_Range  : Ranges;
        end record;
    type Items is array (Positive range <>) of Node;
    type Map(The_Size : Positive) is
        record
            The_Items : Items(1 .. The_Size);
            The_Count : Natural := 0;
        end record;
end Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator
is

    procedure Find (The_Domain : in     Domain;
                    In_The_Map : in  Map;
                    The_Bucket : out Natural) is
        Initial_Probe     : Natural :=
                            Hash_Of(The_Domain) mod
In_The_Map.The_Size;
        Temporary_Index  : Positive;
        Temporary_Bucket : Natural;
    begin
        Temporary_Bucket := 0;
        for Index in In_The_Map.The_Items'Range loop
            Temporary_Index :=
              ((Index + Initial_Probe - 2) mod In_The_Map.The_Size) +
1;
            case In_The_Map.The_Items(Temporary_Index).The_State is
                when Empty   =>
                    if Temporary_Bucket = 0 then
                        Temporary_Bucket := Temporary_Index;
                    end if;
                    The_Bucket := Temporary_Bucket;
                    return;
                when Deleted =>
                    if Temporary_Bucket = 0 then
                        Temporary_Bucket := Temporary_Index;
                    end if;
                when Bound   =>
                    if
In_The_Map.The_Items(Temporary_Index).The_Domain =
                       The_Domain then
                       The_Bucket := Temporary_Index;
                       return;
                    end if;
            end case;
        end loop;
        The_Bucket := Temporary_Bucket;
    end Find;

    procedure Copy (From_The_Map : in     Map;
                    To_The_Map   : in out Map) is
        The_Bucket : Natural;
    begin
        if From_The_Map.The_Count > To_The_Map.The_Size then
            raise Overflow;
        else
            for Index in To_The_Map.The_Items'Range loop
                To_The_Map.The_Items(Index).The_State := Empty;
            end loop;
            To_The_Map.The_Count := 0;
            for Index in From_The_Map.The_Items'Range loop
                if From_The_Map.The_Items(Index).The_State = Bound
then
                    Find(From_The_Map.The_Items(Index).The_Domain,
                         To_The_Map, The_Bucket);
                    To_The_Map.The_Items(The_Bucket) :=
                      From_The_Map.The_Items(Index);
                end if;
            end loop;
            To_The_Map.The_Count := From_The_Map.The_Count;
        end if;
    end Copy;

    procedure Clear (The_Map : in out Map) is
    begin
        for Index in The_Map.The_Items'Range loop
            The_Map.The_Items(Index).The_State := Empty;
        end loop;
        The_Map.The_Count := 0;
    end Clear;

    procedure Bind (The_Domain    : in     Domain;
                    And_The_Range : in     Ranges;
                    In_The_Map    : in out Map) is
        The_Bucket : Natural;
    begin
        Find(The_Domain, In_The_Map, The_Bucket);
        if In_The_Map.The_Items(The_Bucket).The_State = Bound then
            raise Multiple_Binding;
        else
            In_The_Map.The_Items(The_Bucket) :=
              Node'(Bound, The_Domain, And_The_Range);
            In_The_Map.The_Count := In_The_Map.The_Count + 1;
        end if;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Bind;

    procedure Unbind (The_Domain : in     Domain;
                      In_The_Map : in out Map) is
        The_Bucket : Natural;
    begin
        Find(The_Domain, In_The_Map, The_Bucket);
        if In_The_Map.The_Items(The_Bucket).The_State = Bound then
            In_The_Map.The_Items(The_Bucket).The_State := Deleted;
            In_The_Map.The_Count := In_The_Map.The_Count - 1;
        else
            raise Domain_Is_Not_Bound;
        end if;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Unbind;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left   : in Map;
                        Right  : in Map;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Map : in Map;
                         Result  : out Natural) is
    begin
        Result := Extent_Of(The_Map);
    end Extent_Of;

    procedure Is_Empty (The_Map : in Map;
                        Result  : out Boolean) is
    begin
        Result := Is_Empty(The_Map);
    end Is_Empty;

    procedure Is_Bound (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Boolean) is
    begin
        Result := Is_Bound(The_Domain,In_The_Map);
    end Is_Bound;

    procedure Range_Of (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Ranges) is
    begin
        Result := Range_Of(The_Domain,In_The_Map);
    end Range_Of;

--  end of modification

    function Is_Equal (Left  : in Map;
                       Right : in Map) return Boolean is
        Temporary_Index : Natural;
    begin
        if Left.The_Count /= Right.The_Count then
            return False;
        else
            for Index in 1 .. Left.The_Size loop
                if Left.The_Items(Index).The_State = Bound then
                    Temporary_Index := 0;
                    for Inner_Index in 1 .. Right.The_Size loop
                        if (Right.The_Items(Index).The_State = Bound)
and then
                           (Left.The_Items(Index).The_Domain =
                            Right.The_Items(Inner_Index).The_Domain)
then
                            Temporary_Index := Inner_Index;
                            exit;
                        end if;
                    end loop;
                    if Left.The_Items(Index).The_Range /=
                       Right.The_Items(Temporary_Index).The_Range then
                        return False;
                    end if;
                end if;
            end loop;
            return True;
        end if;
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Extent_Of (The_Map : in Map) return Natural is
    begin
        return The_Map.The_Count;
    end Extent_Of;
```

```
function Is_Empty (The_Map : in Map) return Boolean is
begin
    return (The_Map.The_Count = 0);
end Is_Empty;

function Is_Bound (The_Domain : in Domain;
                   In_The_Map : in Map) return Boolean is
    The_Bucket : Natural;
begin
    Find(The_Domain, In_The_Map, The_Bucket);
    return (In_The_Map.The_Items(The_Bucket).The_State = Bound);
exception
    when Constraint_Error =>
        return False;
end Is_Bound;

function Range_Of (The_Domain : in Domain;
                   In_The_Map : in Map) return Ranges is
    The_Bucket : Natural;
begin
    Find(The_Domain, In_The_Map, The_Bucket);
    if In_The_Map.The_Items(The_Bucket).The_State = Bound then
            return In_The_Map.The_Items(The_Bucket).The_Range;
        else
            raise Domain_Is_Not_Bound;
        end if;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Range_Of;

    procedure Iterate (Over_The_Map : in Map) is
        Continue : Boolean;
    begin
        for Index in Over_The_Map.The_Items'Range loop
            if Over_The_Map.The_Items(Index).The_State = Bound then
                Process(Over_The_Map.The_Items(Index).The_Domain,
                        Over_The_Map.The_Items(Index).The_Range,
                        Continue);
                exit when not Continue;
            end if;
        end loop;
    end Iterate;

end Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Domain : PRIVATE_TYPE,
    Ranges : PRIVATE_TYPE,
    Hash_Of : FUNCTION[The_Domain : Domain, RETURN : Positive],
    Hash_Of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positive]]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Map : Map,
      To_The_Map : Map
    OUTPUT
      To_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Bind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      And_The_Range : Ranges,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Unbind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Map,
      Right : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Bound
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Range_Of
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Ranges
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Domain : in[t : Domain], The_Range :
in[t : Ranges], Continue : out[t : Boolean]]
    INPUT
      Over_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Bounded_Managed_Iterator
END
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Domain is private;
    type Ranges is private;
    Number_Of_Buckets : in Positive;
    with function Hash_Of (The_Domain : in Domain) return Positive;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    with procedure Hash_Of (The_Domain : in Domain;
                            Result     : out Positive);
--  end of modication

package Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator
is

    type Map is limited private;

    procedure Copy    (From_The_Map  : in     Map;
                       To_The_Map    : in out Map);
    procedure Clear   (The_Map       : in out Map);
    procedure Bind    (The_Domain    : in     Domain;
                       And_The_Range : in     Ranges;
                       In_The_Map    : in out Map);
    procedure Unbind  (The_Domain    : in     Domain;
                       In_The_Map    : in out Map);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal  (Left   : in Map;
                         Right  : in Map;
                         Result : out Boolean);
    procedure Extent_Of (The_Map   : in Map;
                         Result    : out Natural);
    procedure Is_Empty  (The_Map   : in Map;
                              Result    : out Boolean);
    procedure Is_Bound  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Boolean);
    procedure Range_Of  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Ranges);
--  end of modication

    function Is_Equal  (Left       : in Map;
                        Right      : in Map) return Boolean;
    function Extent_Of (The_Map    : in Map) return Natural;
    function Is_Empty  (The_Map    : in Map) return Boolean;
    function Is_Bound  (The_Domain : in Domain;
                        In_The_Map : in Map) return Boolean;
    function Range_Of  (The_Domain : in Domain;
                        In_The_Map : in Map) return Ranges;

    Overflow           : exception;
    Domain_Is_Not_Bound : exception;
    Multiple_Binding   : exception;

private
    type Node;
    type Structure is access Node;
    type Map is array (Positive range 1 .. Number_Of_Buckets) of
Structure;
end Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--         "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator is

    type Node is
        record
            The_Domain : Domain;
            The_Range  : Ranges;
            Next       : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Structure,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Find (The_Domain    : in     Domain;
                    In_The_Map    : in     Map;
                    The_Bucket    : out    Positive;
                    Previous_Node : in out Structure;
                    Current_Node  : in out Structure) is
       Temporary_Bucket : Positive :=
                    (Hash_Of (The_Domain) mod
Number_Of_Buckets) + 1;
    begin
        The_Bucket := Temporary_Bucket;
        Current_Node := In_The_Map(Temporary_Bucket);
        while Current_Node /= null loop
            if Current_Node.The_Domain = The_Domain then
                return;
            else
                Previous_Node := Current_Node;
                Current_Node := Current_Node.Next;
            end if;
        end loop;
    end Find;

    procedure Copy (From_The_Map : in     Map;
                    To_The_Map   : in out Map) is
        From_Index : Structure;
        To_Index   : Structure;
    begin
        for Index in To_The_Map'Range loop
            Node_Manager.Free(To_The_Map(Index));
        end loop;
        for Index in From_The_Map'Range loop
            From_Index := From_The_Map(Index);
            if From_The_Map(Index) /= null then
                To_The_Map(Index) := Node_Manager.New_Item;
                To_The_Map(Index).The_Domain := From_Index.The_Domain;
                To_The_Map(Index).The_Range := From_Index.The_Range;
                To_Index := To_The_Map(Index);
                From_Index := From_Index.Next;
                while From_Index /= null loop
                    To_Index.Next := Node_Manager.New_Item;
                    To_Index.Next.The_Domain := From_Index.The_Domain;
                    To_Index.Next.The_Range := From_Index.The_Range;
                    To_Index := To_Index.Next;
                    From_Index := From_Index.Next;
                end loop;
            end if;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Map : in out Map) is
    begin
        for Index in The_Map'Range loop
            Node_Manager.Free(The_Map(Index));
        end loop;
    end Clear;

    procedure Bind (The_Domain    : in     Domain;
                    And_The_Range : in     Ranges;
                    In_The_Map    : in out Map) is
        The_Bucket     : Positive;
        Previous_Node  : Structure;
        Current_Node   : Structure;
        Temporary_Node : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Current_Node /= null then
            raise Multiple_Binding;
        else
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Domain := The_Domain;
            Temporary_Node.The_Range := And_The_Range;
            Temporary_Node.Next := In_The_Map(The_Bucket);
            In_The_Map(The_Bucket) := Temporary_Node;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Bind;

    procedure Unbind (The_Domain : in     Domain;
                      In_The_Map : in out Map) is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Previous_Node = null then
            In_The_Map (The_Bucket) := Current_Node.Next;
        else
            Previous_Node.Next := Current_Node.Next;
        end if;
        Current_Node.Next := null;
        Node_Manager.Free(Current_Node);
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Unbind;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left   : in Map;
                        Right  : in Map;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Map : in Map;
                         Result  : out Natural) is
    begin
        Result := Extent_Of(The_Map);
    end Extent_Of;

    procedure Is_Empty (The_Map : in Map;
                        Result  : out Boolean) is
    begin
        Result := Is_Empty(The_Map);
    end Is_Empty;

    procedure Is_Bound (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Boolean) is
    begin
        Result := Is_Bound(The_Domain,In_The_Map);
    end Is_Bound;

    procedure Range_Of (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Ranges) is
    begin
        Result := Range_Of(The_Domain,In_The_Map);
    end Range_Of;

--  end of modification

    function Is_Equal (Left  : in Map;
                       Right : in Map) return Boolean is
        Left_Index  : Structure;
        Right_Index : Structure;
        Left_Count  : Natural;
        Right_Count : Natural;
    begin
```

```
        for Index in Left'Range loop
            if (Left(Index) = null) xor (Right(Index) = null) then
                return False;
            else
                Left_Index := Left(Index);
                Left_Count := 0;
                while Left_Index /= null loop
                    Right_Index := Right(Index);
                    while Right_Index /= null loop
                        if (Left_Index.The_Domain =
                            Right_Index.The_Domain) then
                            exit;
                        else
                            Right_Index := Right_Index.Next;
                        end if;
                    end loop;
                    if Left_Index.The_Range /= Right_Index.The_Range
then
                        return False;
                    else
                        Left_Index := Left_Index.Next;
                        Left_Count := Left_Count + 1;
                    end if;
                end loop;
                Right_Index := Right(Index);
                Right_Count := 0;
                while Right_Index /= null loop
                    Right_Index := Right_Index.Next;
                    Right_Count := Right_Count + 1;
                end loop;
                if Left_Count /= Right_Count then
                    return False;
                end if;
            end if;
        end loop;
        return True;
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Extent_Of (The_Map : in Map) return Natural is
        Count          : Natural    := 0;
        Temporary_Node : Structure;
    begin
        for Index in The_Map'Range loop
            Temporary_Node := The_Map(Index);
            while Temporary_Node /= null loop
                Count := Count + 1;
                Temporary_Node := Temporary_Node.Next;
            end loop;
        end loop;
        return Count;
    end Extent_Of;

    function Is_Empty (The_Map : in Map) return Boolean is
    begin
        return (The_Map = Map'(others => null));
    end Is_Empty;

    function Is_Bound (The_Domain : in Domain;
                       In_The_Map : in Map) return Boolean is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return (Current_Node /= null);
    end Is_Bound;

    function Range_Of (The_Domain : in Domain;
                       In_The_Map : in Map) return Ranges is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return Current_Node.The_Range;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Range_Of;

end Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL BOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Domain : PRIVATE_TYPE,
    Ranges : PRIVATE_TYPE,
    Hash_Of : FUNCTION[The_Domain : Domain, RETURN : Positive],
    Hash_Of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positive]]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Map : Map,
      To_The_Map : Map
    OUTPUT
      To_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Bind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      And_The_Range : Ranges,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Unbind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Map,
      Right : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Bound
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Range_Of
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Ranges
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Unbounded_Managed_Noniterator
END
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Domain is private;
    type Ranges is private;
    Number_Of_Buckets : in Positive;
    with function Hash_Of (The_Domain : in Domain) return Positive;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    with procedure Hash_Of (The_Domain : in Domain;
                            Result     : out Positive);
--  end of modication

package Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator is

    type Map is limited private;

    procedure Copy   (From_The_Map   : in     Map;
                      To_The_Map     : in out Map);
    procedure Clear  (The_Map        : in out Map);
    procedure Bind   (The_Domain     : in     Domain;
                      And_The_Range  : in     Ranges;
                      In_The_Map     : in out Map);
    procedure Unbind (The_Domain     : in     Domain;
                      In_The_Map     : in out Map);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal  (Left   : in Map;
                         Right  : in Map;
                         Result : out Boolean);
    procedure Extent_Of (The_Map   : in Map;
                         Result    : out Natural);
    procedure Is_Empty  (The_Map   : in Map;
                                   Result     : out Boolean);
    procedure Is_Bound  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Boolean);
    procedure Range_Of  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Ranges);
--  end of modication

    function Is_Equal  (Left       : in Map;
                        Right      : in Map) return Boolean;
    function Extent_Of (The_Map    : in Map) return Natural;
    function Is_Empty  (The_Map    : in Map) return Boolean;
    function Is_Bound  (The_Domain : in Domain;
                        In_The_Map : in Map) return Boolean;
    function Range_Of  (The_Domain : in Domain;
                        In_The_Map : in Map) return Ranges;

    generic
        with procedure Process (The_Domain : in  Domain;
                                The_Range  : in  Ranges;
                                Continue   : out Boolean);
    procedure Iterate (Over_The_Map : in Map);

    Overflow            : exception;
    Domain_Is_Not_Bound : exception;
    Multiple_Binding    : exception;

private
    type Node;
    type Structure is access Node;
    type Map is array (Positive range 1 .. Number_Of_Buckets) of
Structure;
end Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator is

    type Node is
        record
            The_Domain : Domain;
            The_Range  : Ranges;
            Next       : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                 (Item        => Node,
                                  Pointer     => Structure,
                                  Free        => Free,
                                  Set_Pointer => Set_Next,
                                  Pointer_Of  => Next_Of);

    procedure Find (The_Domain    : in     Domain;
                    In_The_Map    : in     Map;
                    The_Bucket    : out    Positive;
                    Previous_Node : in out Structure;
                    Current_Node  : in out Structure) is
      Temporary_Bucket : Positive :=
                    (Hash_Of (The_Domain) mod
Number_Of_Buckets) + 1;
    begin
        The_Bucket := Temporary_Bucket;
        Current_Node := In_The_Map(Temporary_Bucket);
        while Current_Node /= null loop
            if Current_Node.The_Domain = The_Domain then
                return;
            else
                Previous_Node := Current_Node;
                Current_Node := Current_Node.Next;
            end if;
        end loop;
    end Find;

    procedure Copy (From_The_Map : in     Map;
                    To_The_Map   : in out Map) is
        From_Index : Structure;
        To_Index   : Structure;
    begin
        for Index in To_The_Map'Range loop
            Node_Manager.Free(To_The_Map(Index));
        end loop;
        for Index in From_The_Map'Range loop
            From_Index := From_The_Map(Index);
            if From_The_Map(Index) /= null then
                To_The_Map(Index) := Node_Manager.New_Item;
                To_The_Map(Index).The_Domain := From_Index.The_Domain;
                To_The_Map(Index).The_Range := From_Index.The_Range;
                To_Index := To_The_Map(Index);
                From_Index := From_Index.Next;
                while From_Index /= null loop
                    To_Index.Next := Node_Manager.New_Item;
                    To_Index.Next.The_Domain := From_Index.The_Domain;
                    To_Index.Next.The_Range := From_Index.The_Range;
                    To_Index := To_Index.Next;
                    From_Index := From_Index.Next;
                end loop;
            end if;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Map : in out Map) is
    begin
        for Index in The_Map'Range loop
            Node_Manager.Free(The_Map(Index));
        end loop;
    end Clear;

    procedure Bind (The_Domain    : in     Domain;
                    And_The_Range : in     Ranges;
                    In_The_Map    : in out Map) is
        The_Bucket     : Positive;
        Previous_Node  : Structure;
        Current_Node   : Structure;
        Temporary_Node : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Current_Node /= null then
            raise Multiple_Binding;
        else
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Domain := The_Domain;
            Temporary_Node.The_Range := And_The_Range;
            Temporary_Node.Next := In_The_Map(The_Bucket);
            In_The_Map(The_Bucket) := Temporary_Node;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Bind;

    procedure Unbind (The_Domain : in     Domain;
                      In_The_Map : in out Map) is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Previous_Node = null then
            In_The_Map (The_Bucket) := Current_Node.Next;
        else
            Previous_Node.Next := Current_Node.Next;
        end if;
        Current_Node.Next := null;
        Node_Manager.Free(Current_Node);
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Unbind;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left   : in Map;
                        Right  : in Map;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Map : in Map;
                         Result  : out Natural) is
    begin
        Result := Extent_Of(The_Map);
    end Extent_Of;

    procedure Is_Empty (The_Map : in Map;
                        Result  : out Boolean) is
    begin
        Result := Is_Empty(The_Map);
    end Is_Empty;

    procedure Is_Bound (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Boolean) is
    begin
        Result := Is_Bound(The_Domain,In_The_Map);
    end Is_Bound;

    procedure Range_Of (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Ranges) is
    begin
        Result := Range_Of(The_Domain,In_The_Map);
    end Range_Of;

--  end of modification

    function Is_Equal (Left  : in Map;
                       Right : in Map) return Boolean is
        Left_Index  : Structure;
        Right_Index : Structure;
        Left_Count  : Natural;
        Right_Count : Natural;
    begin
```

```
        for Index in Left'Range loop
            if (Left(Index) = null) xor (Right(Index) = null) then
                return False;
            else
                Left_Index := Left(Index);
                Left_Count := 0;
                while Left_Index /= null loop
                    Right_Index := Right(Index);
                    while Right_Index /= null loop
                        if (Left_Index.The_Domain =
                            Right_Index.The_Domain) then
                            exit;
                        else
                            Right_Index := Right_Index.Next;
                        end if;
                    end loop;
                    if Left_Index.The_Range /= Right_Index.The_Range
then
                        return False;
                    else
                        Left_Index := Left_Index.Next;
                        Left_Count := Left_Count + 1;
                    end if;
                end loop;
                Right_Index := Right(Index);
                Right_Count := 0;
                while Right_Index /= null loop
                    Right_Index := Right_Index.Next;
                    Right_Count := Right_Count + 1;
                end loop;
                if Left_Count /= Right_Count then
                    return False;
                end if;
            end if;
        end loop;
        return True;
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Extent_Of (The_Map : in Map) return Natural is
        Count          : Natural   := 0;
        Temporary_Node : Structure;
    begin
        for Index in The_Map'Range loop
            Temporary_Node := The_Map(Index);
            while Temporary_Node /= null loop
                Count := Count + 1;
                Temporary_Node := Temporary_Node.Next;
            end loop;
        end loop;
        return Count;
    end Extent_Of;

    function Is_Empty (The_Map : in Map) return Boolean is
    begin
        return (The_Map = Map'(others => null));
    end Is_Empty;

    function Is_Bound (The_Domain : in Domain;
                       In_The_Map : in Map) return Boolean is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return (Current_Node /= null);
    end Is_Bound;

    function Range_Of (The_Domain : in Domain;
                       In_The_Map : in Map) return Ranges is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return Current_Node.The_Range;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Range_Of;

    procedure Iterate (Over_The_Map : in Map) is
        The_Bucket : Positive := Over_The_Map'Last;
        The_Node   : Structure;
        Continue   : Boolean;
    begin
        for The_Iterator in Over_The_Map'Range loop
            if Over_The_Map(The_Iterator) /= null then
                The_Bucket := The_Iterator;
                The_Node := Over_The_Map(The_Iterator);
                exit;
            end if;
        end loop;
        while The_Node /= null loop
            Process(The_Node.The_Domain, The_Node.The_Range,
Continue);
            exit when not Continue;
            The_Node := The_Node.Next;
            if The_Node = null then
                for The_Iterator in (The_Bucket + 1) ..
Over_The_Map'Last loop
                    if Over_The_Map(The_Iterator) /= null then
                        The_Bucket := The_Iterator;
                        The_Node := Over_The_Map(The_Iterator);
                        exit;
                    end if;
                end loop;
            end if;
        end loop;
    end Iterate;

end Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Domain : PRIVATE_TYPE,
    Ranges : PRIVATE_TYPE,
    Hash_Of : FUNCTION[The_Domain : Domain, RETURN : Positive],
    Hash_Of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positive]]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Map : Map,
      To_The_Map : Map
    OUTPUT
      To_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Bind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      And_The_Range : Ranges,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Unbind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Map,
      Right : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Bound
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Range_Of
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Ranges
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Domain : in[t : Domain], The_Range :
in[t : Ranges], Continue : out[t : Boolean]]
    INPUT
      Over_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
END
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Domain is private;
    type Ranges is private;
    Number_Of_Buckets : in Positive;
    with function Hash_Of (The_Domain : in Domain) return Positive;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    with procedure Hash_Of (The_Domain : in Domain;
                            Result     : out Positive);
--  end of modication

package
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator is

    type Map is limited private;

    procedure Copy    (From_The_Map  : in     Map;
                       To_The_Map    : in out Map);
    procedure Clear   (The_Map       : in out Map);
    procedure Bind    (The_Domain    : in     Domain;
                       And_The_Range : in     Ranges;
                       In_The_Map    : in out Map);
    procedure Unbind  (The_Domain    : in     Domain;
                       In_The_Map    : in out Map);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal  (Left   : in Map;
                         Right  : in Map;
                         Result : out Boolean);
    procedure Extent_Of (The_Map    : in Map;
                         Result     : out Natural);
    procedure Is_Empty  (The_Map    : in Map;
                                Result     : out Boolean);
    procedure Is_Bound  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Boolean);
    procedure Range_Of  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Ranges);
--  end of modication

    function Is_Equal  (Left       : in Map;
                        Right      : in Map) return Boolean;
    function Extent_Of (The_Map    : in Map) return Natural;
    function Is_Empty  (The_Map    : in Map) return Boolean;
    function Is_Bound  (The_Domain : in Domain;
                        In_The_Map : in Map) return Boolean;
    function Range_Of  (The_Domain : in Domain;
                        In_The_Map : in Map) return Ranges;

    Overflow            : exception;
    Domain_Is_Not_Bound : exception;
    Multiple_Binding    : exception;

private
    type Node;
    type Structure is access Node;
    type Map is array (Positive range 1 .. Number_Of_Buckets) of
Structure;
end Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator is

    type Node is
        record
            The_Domain : Domain;
            The_Range  : Ranges;
            Next       : Structure;
        end record;

    procedure Find (The_Domain    : in     Domain;
                    In_The_Map    : in     Map;
                    The_Bucket    : out    Positive;
                    Previous_Node : in out Structure;
                    Current_Node  : in out Structure) is
       Temporary_Bucket : Positive :=
                 (Hash_Of (The_Domain) mod
Number_Of_Buckets) + 1;
    begin
        The_Bucket := Temporary_Bucket;
        Current_Node := In_The_Map(Temporary_Bucket);
        while Current_Node /= null loop
            if Current_Node.The_Domain = The_Domain then
                return;
            else
                Previous_Node := Current_Node;
                Current_Node := Current_Node.Next;
            end if;
        end loop;
    end Find;

    procedure Copy (From_The_Map : in     Map;
                    To_The_Map   : in out Map) is
        From_Index : Structure;
        To_Index   : Structure;
    begin
        for Index in From_The_Map'Range loop
            From_Index := From_The_Map(Index);
            if From_The_Map(Index) = null then
                To_The_Map(Index) := null;
            else
                To_The_Map(Index) := new Node'
                                      (The_Domain =>
From_Index.The_Domain,
                                       The_Range  =>
From_Index.The_Range,
                                       Next       => null);
                To_Index := To_The_Map(Index);
                From_Index := From_Index.Next;
                while From_Index /= null loop
                    To_Index.Next := new Node'
                                      (The_Domain =>
From_Index.The_Domain,
                                       The_Range  =>
From_Index.The_Range,
                                       Next       => null);
                    To_Index := To_Index.Next;
                    From_Index := From_Index.Next;
                end loop;
            end if;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Map : in out Map) is
    begin
        The_Map := Map'(others => null);
    end Clear;

    procedure Bind (The_Domain    : in     Domain;
                    And_The_Range : in     Ranges;
                    In_The_Map    : in out Map) is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Current_Node /= null then
            raise Multiple_Binding;
        else
            In_The_Map(The_Bucket) := new Node'
                                      (The_Domain => The_Domain,
                                       The_Range  => And_The_Range,
                                       Next       =>
In_The_Map(The_Bucket));
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Bind;

    procedure Unbind (The_Domain : in     Domain;
                      In_The_Map : in out Map) is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Previous_Node = null then
            In_The_Map (The_Bucket) := Current_Node.Next;
        else
            Previous_Node.Next := Current_Node.Next;
        end if;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Unbind;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left   : in Map;
                        Right  : in Map;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Map : in Map;
                         Result  : out Natural) is
    begin
        Result := Extent_Of(The_Map);
    end Extent_Of;

    procedure Is_Empty (The_Map : in Map;
                        Result  : out Boolean) is
    begin
        Result := Is_Empty(The_Map);
    end Is_Empty;

    procedure Is_Bound (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Boolean) is
    begin
        Result := Is_Bound(The_Domain,In_The_Map);
    end Is_Bound;

    procedure Range_Of (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Ranges) is
    begin
        Result := Range_Of(The_Domain,In_The_Map);
    end Range_Of;

--  end of modification

    function Is_Equal (Left  : in Map;
                       Right : in Map) return Boolean is
        Left_Index  : Structure;
        Right_Index : Structure;
        Left_Count  : Natural;
        Right_Count : Natural;
    begin
        for Index in Left'Range loop
            if (Left(Index) = null) xor (Right(Index) = null) then
                return False;
            else
                Left_Index := Left(Index);
                Left_Count := 0;
                while Left_Index /= null loop
                    Right_Index := Right(Index);
                    while Right_Index /= null loop
                        if (Left_Index.The_Domain =
                            Right_Index.The_Domain) then
                            exit;
                        else
                            Right_Index := Right_Index.Next;
                        end if;
                    end loop;
                    if Left_Index.The_Range /= Right_Index.The_Range
then
                        return False;
                    else
                        Left_Index := Left_Index.Next;
                        Left_Count := Left_Count + 1;
                    end if;
                end loop;
```

```
            Right_Index := Right(Index);
            Right_Count := 0;
            while Right_Index /= null loop
                Right_Index := Right_Index.Next;
                Right_Count := Right_Count + 1;
            end loop;
            if Left_Count /= Right_Count then
                return False;
            end if;
        end if;
    end loop;
    return True;
exception
    when Constraint_Error =>
        return False;
end Is_Equal;

function Extent_Of (The_Map : in Map) return Natural is
    Count          : Natural   := 0;
    Temporary_Node : Structure;
begin
    for Index in The_Map'Range loop
        Temporary_Node := The_Map(Index);
        while Temporary_Node /= null loop
            Count := Count + 1;
            Temporary_Node := Temporary_Node.Next;
        end loop;
    end loop;
    return Count;
end Extent_Of;
```

```
    function Is_Empty (The_Map : in Map) return Boolean is
    begin
        return (The_Map = Map'(others => null));
    end Is_Empty;

    function Is_Bound (The_Domain : in Domain;
                       In_The_Map : in Map) return Boolean is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return (Current_Node /= null);
    end Is_Bound;

    function Range_Of (The_Domain : in Domain;
                       In_The_Map : in Map) return Ranges is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return Current_Node.The_Range;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Range_Of;

end Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Domain : PRIVATE_TYPE,
    Ranges : PRIVATE_TYPE,
    Hash_Of : FUNCTION[The_Domain : Domain, RETURN : Positive],
    Hash_Of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positive]]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Map : Map,
      To_The_Map : Map
    OUTPUT
      To_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Bind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      And_The_Range : Ranges,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Unbind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Map,
      Right : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Bound
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Range_Of
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Ranges
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Noniterator
END
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Domain is private;
    type Ranges is private;
    Number_Of_Buckets : in Positive;
    with function Hash_Of (The_Domain : in Domain) return Positive;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    with procedure Hash_Of (The_Domain : in Domain;
                            Result     : out Positive);
--  end of modication

package Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
is

    type Map is limited private;

    procedure Copy   (From_The_Map   : in      Map;
                      To_The_Map     : in out Map);
    procedure Clear  (The_Map        : in out Map);
    procedure Bind   (The_Domain     : in      Domain;
                      And_The_Range  : in      Ranges;
                      In_The_Map     : in out Map);
    procedure Unbind (The_Domain     : in      Domain;
                      In_The_Map     : in out Map);

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal  (Left   : in Map;
                         Right  : in Map;
                         Result : out Boolean);
    procedure Extent_Of (The_Map    : in Map;
                         Result     : out Natural);
    procedure Is_Empty  (The_Map    : in Map;
                               Result     : out Boolean);
    procedure Is_Bound  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Boolean);
    procedure Range_Of  (The_Domain : in Domain;
                         In_The_Map : in Map;
                         Result     : out Ranges);
--  end of modication

    function Is_Equal  (Left       : in Map;
                        Right      : in Map) return Boolean;
    function Extent_Of (The_Map    : in Map) return Natural;
    function Is_Empty  (The_Map    : in Map) return Boolean;
    function Is_Bound  (The_Domain : in Domain;
                        In_The_Map : in Map) return Boolean;
    function Range_Of  (The_Domain : in Domain;
                        In_The_Map : in Map) return Ranges;

    generic
        with procedure Process (The_Domain : in  Domain;
                                The_Range  : in  Ranges;
                                Continue   : out Boolean);
    procedure Iterate (Over_The_Map : in Map);

    Overflow            : exception;
    Domain_Is_Not_Bound : exception;
    Multiple_Binding    : exception;

private
    type Node;
    type Structure is access Node;
    type Map is array (Positive range 1 .. Number_Of_Buckets) of
Structure;
end Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator is

    type Node is
        record
            The_Domain : Domain;
            The_Range  : Ranges;
            Next       : Structure;
        end record;

    procedure Find (The_Domain    : in     Domain;
                    In_The_Map    : in     Map;
                    The_Bucket    : out    Positive;
                    Previous_Node : in out Structure;
                    Current_Node  : in out Structure) is
      Temporary_Bucket : Positive :=
                         (Hash_Of (The_Domain) mod
Number_Of_Buckets) + 1;
    begin
        The_Bucket := Temporary_Bucket;
        Current_Node := In_The_Map(Temporary_Bucket);
        while Current_Node /= null loop
            if Current_Node.The_Domain = The_Domain then
                return;
            else
                Previous_Node := Current_Node;
                Current_Node := Current_Node.Next;
            end if;
        end loop;
    end Find;

    procedure Copy (From_The_Map : in     Map;
                    To_The_Map   : in out Map) is
        From_Index : Structure;
        To_Index   : Structure;
    begin
        for Index in From_The_Map'Range loop
            From_Index := From_The_Map(Index);
            if From_The_Map(Index) = null then
                To_The_Map(Index) := null;
            else
                To_The_Map(Index) := new Node'
                                    (The_Domain =>
From_Index.The_Domain,
                                     The_Range  =>
From_Index.The_Range,
                                     Next       => null);
                To_Index := To_The_Map(Index);
                From_Index := From_Index.Next;
                while From_Index /= null loop
                    To_Index.Next := new Node'
                                    (The_Domain =>
From_Index.The_Domain,
                                     The_Range  =>
From_Index.The_Range,
                                     Next       => null);
                    To_Index := To_Index.Next;
                    From_Index := From_Index.Next;
                end loop;
            end if;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Map : in out Map) is
    begin
        The_Map := Map'(others => null);
    end Clear;

    procedure Bind (The_Domain    : in     Domain;
                    And_The_Range : in     Ranges;
                    In_The_Map    : in out Map) is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Current_Node /= null then
            raise Multiple_Binding;
        else
            In_The_Map(The_Bucket) := new Node'
                                    (The_Domain => The_Domain,
                                     The_Range  => And_The_Range,
                                     Next       =>
In_The_Map(The_Bucket));
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Bind;

    procedure Unbind (The_Domain : in     Domain;
                      In_The_Map : in out Map) is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        if Previous_Node = null then
            In_The_Map (The_Bucket) := Current_Node.Next;
        else
            Previous_Node.Next := Current_Node.Next;
        end if;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Unbind;

--  modified by Tuan Nguyen and Vincent Hong
--  date:  8 April 1995
--  adding procedures to replace functions

    procedure Is_Equal (Left   : in Map;
                        Right  : in Map;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Map : in Map;
                         Result  : out Natural) is
    begin
        Result := Extent_Of(The_Map);
    end Extent_Of;

    procedure Is_Empty (The_Map : in Map;
                        Result  : out Boolean) is
    begin
        Result := Is_Empty(The_Map);
    end Is_Empty;

    procedure Is_Bound (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Boolean) is
    begin
        Result := Is_Bound(The_Domain,In_The_Map);
    end Is_Bound;

    procedure Range_Of (The_Domain : in Domain;
                        In_The_Map : in Map;
                        Result     : out Ranges) is
    begin
        Result := Range_Of(The_Domain,In_The_Map);
    end Range_Of;

--  end of modification

    function Is_Equal (Left  : in Map;
                       Right : in Map) return Boolean is
        Left_Index  : Structure;
        Right_Index : Structure;
        Left_Count  : Natural;
        Right_Count : Natural;
    begin
        for Index in Left'Range loop
            if (Left(Index) = null) xor (Right(Index) = null) then
                return False;
            else
                Left_Index := Left(Index);
                Left_Count := 0;
                while Left_Index /= null loop
                    Right_Index := Right(Index);
                    while Right_Index /= null loop
                        if (Left_Index.The_Domain =
                            Right_Index.The_Domain) then
                            exit;
                        else
                            Right_Index := Right_Index.Next;
                        end if;
                    end loop;
                    if Left_Index.The_Range /= Right_Index.The_Range
then
                        return False;
                    else
                        Left_Index := Left_Index.Next;
                        Left_Count := Left_Count + 1;
                    end if;
                end loop;
```

```ada
                Right_Index := Right(Index);
                Right_Count := 0;
                while Right_Index /= null loop
                    Right_Index := Right_Index.Next;
                    Right_Count := Right_Count + 1;
                end loop;
                if Left_Count /= Right_Count then
                    return False;
                end if;
            end if;
        end loop;
        return True;
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Extent_Of (The_Map : in Map) return Natural is
        Count          : Natural   := 0;
        Temporary_Node : Structure;
    begin
        for Index in The_Map'Range loop
            Temporary_Node := The_Map(Index);
            while Temporary_Node /= null loop
                Count := Count + 1;
                Temporary_Node := Temporary_Node.Next;
            end loop;
        end loop;
        return Count;
    end Extent_Of;

    function Is_Empty (The_Map : in Map) return Boolean is
    begin
        return (The_Map = Map'(others => null));
    end Is_Empty;

    function Is_Bound (The_Domain : in Domain;
                       In_The_Map : in Map) return Boolean is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return (Current_Node /= null);
    end Is_Bound;

    function Range_Of (The_Domain : in Domain;
                       In_The_Map : in Map) return Ranges is
        The_Bucket    : Positive;
        Previous_Node : Structure;
        Current_Node  : Structure;
    begin
        Find(The_Domain, In_The_Map, The_Bucket, Previous_Node,
Current_Node);
        return Current_Node.The_Range;
    exception
        when Constraint_Error =>
            raise Domain_Is_Not_Bound;
    end Range_Of;

    procedure Iterate (Over_The_Map : in Map) is
        The_Bucket : Positive := Over_The_Map'Last;
        The_Node   : Structure;
        Continue   : Boolean;
    begin
        for The_Iterator in Over_The_Map'Range loop
            if Over_The_Map(The_Iterator) /= null then
                The_Bucket := The_Iterator;
                The_Node := Over_The_Map(The_Iterator);
                exit;
            end if;
        end loop;
        while The_Node /= null loop
            Process(The_Node.The_Domain, The_Node.The_Range,
Continue);
            exit when not Continue;
            The_Node := The_Node.Next;
            if The_Node = null then
                for The_Iterator in (The_Bucket + 1) ..
Over_The_Map'Last loop
                    if Over_The_Map(The_Iterator) /= null then
                        The_Bucket := The_Iterator;
                        The_Node := Over_The_Map(The_Iterator);
                        exit;
                    end if;
                end loop;
            end if;
        end loop;
    end Iterate;

end Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator;
```

MAP SIMPLE NONCACHED SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Domain : PRIVATE_TYPE,
    Ranges : PRIVATE_TYPE,
    Hash_Of : FUNCTION[The_Domain : Domain, RETURN : Positive],
    Hash_Of : PROCEDURE[The_Domain : in[t : Domain], Result : out[t :
Positive]]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Map : Map,
      To_The_Map : Map
    OUTPUT
      To_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Bind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      And_The_Range : Ranges,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Unbind
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      In_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Map,
      Right : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Is_Bound
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Range_Of
  SPECIFICATION
    INPUT
      The_Domain : Domain,
      In_The_Map : Map
    OUTPUT
      Result : Ranges
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Domain : in[t : Domain], The_Range :
in[t : Ranges], Continue : out[t : Boolean]]
    INPUT
      Over_The_Map : Map
    EXCEPTIONS
      Overflow, Domain_Is_Not_Bound, Multiple_Binding
  END

END
IMPLEMENTATION ADA
Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
END
```

QUEUES OBJ3 SPECIFICATION

```
obj QUEUE[X :: TRIV] is sort Queue .
   protecting NAT .
   subsorts NzNat < Nat .

*** constructors

   op create      : -> Queue .
   op copy        : Queue Queue -> Queue .
   op clear       : Queue -> Queue .
   op add         : Elt Queue -> Queue .
   op pop         : Queue -> Queue .
   op removeitem : Queue NzNat -> Queue .

*** accessors

   op isequal    : Queue Queue -> Bool .
   op lengthof   : Queue -> Nat .
   op isempty    : Queue -> Bool .
   op frontof    : Queue -> Elt .
   op positionof : Elt Queue -> Nat .

*** exceptions

   op overflow       : -> Queue .
   op underflow      : -> Queue .
   op underflow      : -> Elt .
   op positionerror : -> Nat .

*** variables declarations

   var Q Q1       : Queue .
   var E E1       : Elt .
   var P          : NzNat .

*** axioms

   eq copy(Q,Q1) = Q .

   eq clear(Q) = create .

   eq pop(create) = underflow .
   eq pop(add(E,Q)) = if Q == create then create else add(E,pop(Q)) fi
.

   eq removeitem(create,P) = underflow .
   eq removeitem(add(E,Q),P) = if P == lengthof(Q) + 1 then Q  else
add(E,removeitem(Q,P)) fi .

   eq isequal(Q,Q1) = Q == Q1 .

   eq lengthof(Q) = if Q == create then 0 else 1 + lengthof(pop(Q)) fi
.
   eq isempty(Q) = Q == create .

   eq frontof(create) = underflow .
   eq frontof(add(E,Q)) = if Q == create then E else frontof(Q) fi .

   eq positionof(E,create) = positionerror .
   eq positionof(E,add(E1,Q)) = if E == E1 then lengthof(Q) + 1 else
positionof(E,Q) fi .

endo
```

QUEUES PROFILE CODES

OPERATORS	*SIGNATURES*	*PROFILE CODES*
COPY	A B -> B	3211
CLEAR	A -> A	2201
ADD	A B -> B	3211
POP	A -> A	2201
REMOVE_ITEM	A B -> A	3211
IS_EQUAL	A B -> C	330
LENGTH_OF	A -> B	220
IS_EMPTY	A -> B	220
FRONT_OF	A -> B	220
POSITION_OF	A -> B	220

SET OF PROFILE: {3211,2201,330,220}

QUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATION

```
generic
    type Item is private;
package Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator
is

    type Queue(The_Size : Positive) is limited private;

    procedure Copy        (From_The_Queue  : in     Queue;
                           To_The_Queue    : in out Queue);
    procedure Clear       (The_Queue       : in out Queue);
    procedure Add         (The_Item        : in     Item;
                           To_The_Queue    : in out Queue);
    procedure Pop         (The_Queue       : in out Queue);
    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal    (Left          : in Queue;
                           Right         : in Queue;
                           Result        : out Boolean);
    procedure Length_Of   (The_Queue     : in Queue;
                           Result        : out Natural);
    procedure Is_Empty    (The_Queue     : in Queue;
                           Result        : out Boolean);
    procedure Front_Of    (The_Queue     : in Queue;
                           Result        : Item);
    procedure Position_Of (The_Item      : in Item;
                           In_The_Queue  : in Queue;
                           Result        : out Natural);

--  end of modification

    function Is_Equal    (Left          : in Queue;
                          Right         : in Queue) return Boolean;
    function Length_Of   (The_Queue     : in Queue) return Natural;
    function Is_Empty    (The_Queue     : in Queue) return Boolean;
    function Front_Of    (The_Queue     : in Queue) return Item;
    function Position_Of (The_Item      : in Item;
                          In_The_Queue  : in Queue) return Natural;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Items is array(Positive range <>) of Item;
    type Queue(The_Size : Positive) is
        record
            The_Back  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator is

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
    begin
        if From_The_Queue.The_Back > To_The_Queue.The_Size then
            raise Overflow;
        elsif From_The_Queue.The_Back = 0 then
            To_The_Queue.The_Back := 0;
        else
            To_The_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
              From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
            To_The_Queue.The_Back := From_The_Queue.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue.The_Back := 0;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        To_The_Queue.The_Items(To_The_Queue.The_Back + 1) := The_Item;
        To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        elsif The_Queue.The_Back = 1 then
            The_Queue.The_Back := 0;
        else
            The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
                The_Queue.The_Items(2 .. The_Queue.The_Back);
            The_Queue.The_Back := The_Queue.The_Back - 1;
        end if;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
    begin
        if From_The_Queue.The_Back < At_The_Position then
            raise Position_Error;
        elsif From_The_Queue.The_Back /= At_The_Position then
            From_The_Queue.The_Items
              (At_The_Position .. (From_The_Queue.The_Back - 1)) :=
                From_The_Queue.The_Items
                  ((At_The_Position + 1) .. From_The_Queue.The_Back);
        end if;
        From_The_Queue.The_Back := From_The_Queue.The_Back - 1;
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left          : in Queue;
                             Right         : in Queue;
                             Result        : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of (The_Item     : in Item;
                           In_The_Queue : in Queue;
                           Result       : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
    begin
        if Left.The_Back /= Right.The_Back then
            return False;
        else
            for Index in 1 .. Left.The_Back loop
                if Left.The_Items(Index) /= Right.The_Items(Index)
then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
    begin
        return The_Queue.The_Back;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Back = 0);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        else
            return The_Queue.The_Items(1);
        end if;
    end Front_Of;

    function Position_Of (The_Item     : in Item;
                          In_The_Queue : in Queue) return Natural is
    begin
        for Index in 1 .. In_The_Queue.The_Back loop
            if In_The_Queue.The_Items(Index) = The_Item then
                return Index;
            end if;
        end loop;
        return 0;
    end Position_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        Continue : Boolean;
    begin
        for The_Iterator in 1 .. Over_The_Queue.The_Back loop
            Process(Over_The_Queue.The_Items(The_Iterator), Continue);
            exit when not Continue;
        end loop;
    end Iterate;

end Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Balking_Sequential_Bounded_Managed_Iterator
END
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator is

    type Queue is limited private;

    procedure Copy          (From_The_Queue  : in      Queue;
                             To_The_Queue    : in out Queue);
    procedure Clear         (The_Queue       : in out Queue);
    procedure Add           (The_Item        : in      Item;
                             To_The_Queue    : in out Queue);
    procedure Pop           (The_Queue       : in out Queue);
    procedure Remove_Item   (From_The_Queue  : in out Queue;
                             At_The_Position : in      Positive);

    procedure Is_Equal      (Left           : in Queue;
                             Right          : in Queue;
                             Result         : out Boolean);
    procedure Length_Of     (The_Queue      : in Queue;
                             Result         : out Natural);
    procedure Is_Empty      (The_Queue      : in Queue;
                             Result         : out Boolean);
    procedure Front_Of      (The_Queue      : in Queue;
                             Result         : Item);
    procedure Position_Of   (The_Item       : in Item;
                             In_The_Queue   : in Queue;
                             Result         : out Natural);

--  end of modification

    function Is_Equal    (Left          : in Queue;
                          Right         : in Queue) return Boolean;
    function Length_Of   (The_Queue     : in Queue) return Natural;
    function Is_Empty    (The_Queue     : in Queue) return Boolean;
    function Front_Of    (The_Queue     : in Queue) return Item;
    function Position_Of (The_Item      : in Item;
                          In_The_Queue  : in Queue) return Natural;

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator
  is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                 (Item        => Node,
                                  Pointer     => Structure,
                                  Free        => Free,
                                  Set_Pointer => Set_Next,
                                  Pointer_Of  => Next_Of);

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        Node_Manager.Free(To_The_Queue.The_Front);
        To_The_Queue.The_Back := null;
        if From_The_Queue.The_Front /= null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_The_Queue.The_Front.The_Item := From_Index.The_Item;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.The_Item := From_Index.The_Item;
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        Node_Manager.Free(The_Queue.The_Front);
        The_Queue.The_Back := null;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Front.The_Item := The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            To_The_Queue.The_Back.Next := Node_Manager.New_Item;
            To_The_Queue.The_Back.Next.The_Item:= The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Queue.The_Front;
        The_Queue.The_Front := The_Queue.The_Front.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
        Index.Next := null;
        Node_Manager.Free(Index);
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left         : in Queue;
                             Right        : in Queue;
                             Result       : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_Queue    : in Queue;
                             Result       : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty      (The_Queue    : in Queue;
                             Result       : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of      (The_Queue    : in Queue;
                             Result       : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of   (The_Item     : in Item;
                             In_The_Queue : in Queue;
                             Result       : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
```

```
    Count : Natural   := 0;
    Index : Structure := The_Queue.The_Front;
begin
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Length_Of;

function Is_Empty (The_Queue : in Queue) return Boolean is
begin
    return (The_Queue.The_Front = null);
end Is_Empty;

function Front_Of (The_Queue : in Queue) return Item is
begin
    return The_Queue.The_Front.The_Item;
exception
    when Constraint_Error =>
        raise Underflow;
    end Front_Of;

   function Position_Of (The_Item     : in Item;
                         In_The_Queue : in Queue) return Natural is
        Position : Natural   := 1;
        Index    : Structure := In_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                return Position;
            else
                Position := Position + 1;
                Index := Index.Next;
            end if;
        end loop;
        return 0;
    end Position_Of;

end
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Noniterator
END
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator is

    type Queue(The_Size : Positive) is limited private;

    procedure Copy  (From_The_Queue : in     Queue;
                     To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in     Item;
                     To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean);
    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural);
    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean);
    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item);

--  end of modification

    function Is_Equal  (Left      : in Queue;
                        Right     : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty  (The_Queue : in Queue) return Boolean;
    function Front_Of  (The_Queue : in Queue) return Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow  : exception;
    Underflow : exception;

private
    type Items is array(Positive range <>) of Item;
    type Queue(The_Size : Positive) is
        record
            The_Back  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator
  is

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
    begin
        if From_The_Queue.The_Back > To_The_Queue.The_Size then
            raise Overflow;
        elsif From_The_Queue.The_Back = 0 then
            To_The_Queue.The_Back := 0;
        else
            To_The_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
              From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
            To_The_Queue.The_Back := From_The_Queue.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue.The_Back := 0;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        To_The_Queue.The_Items(To_The_Queue.The_Back + 1) := The_Item;
        To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        elsif The_Queue.The_Back = 1 then
            The_Queue.The_Back := 0;
        else
            The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
                The_Queue.The_Items(2 .. The_Queue.The_Back);
            The_Queue.The_Back := The_Queue.The_Back - 1;
        end if;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left           : in Queue;
                             Right          : in Queue;
                             Result         : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_Queue      : in Queue;
                             Result         : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty      (The_Queue      : in Queue;
                             Result         : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of      (The_Queue      : in Queue;
                             Result         : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
    begin
        if Left.The_Back /= Right.The_Back then
            return False;
        else
            for Index in 1 .. Left.The_Back loop
                if Left.The_Items(Index) /= Right.The_Items(Index)
then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
    begin
        return The_Queue.The_Back;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Back = 0);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        else
            return The_Queue.The_Items(1);
        end if;
    end Front_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        Continue : Boolean;
    begin
        for The_Iterator in 1 .. Over_The_Queue.The_Back loop
            Process(Over_The_Queue.The_Items(The_Iterator), Continue);
            exit when not Continue;
        end loop;
    end Iterate;

end Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Nonbalking_Sequential_Bounded_Managed_Iterator
END
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
  is

    type Queue is limited private;

    procedure Copy  (From_The_Queue : in     Queue;
                     To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in     Item;
                     To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean);
    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural);
    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean);
    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item);

--  end of modification

    function Is_Equal  (Left      : in Queue;
                        Right     : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty  (The_Queue : in Queue) return Boolean;
    function Front_Of  (The_Queue : in Queue) return Item;

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body

Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Structure,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        Node_Manager.Free(To_The_Queue.The_Front);
        To_The_Queue.The_Back := null;
        if From_The_Queue.The_Front /= null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_The_Queue.The_Front.The_Item := From_Index.The_Item;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.The_Item := From_Index.The_Item;
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        Node_Manager.Free(The_Queue.The_Front);
        The_Queue.The_Back := null;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Front.The_Item := The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            To_The_Queue.The_Back.Next := Node_Manager.New_Item;
            To_The_Queue.The_Back.Next.The_Item:= The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Queue.The_Front;
        The_Queue.The_Front := The_Queue.The_Front.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal    (Left        : in Queue;
                           Right       : in Queue;
                           Result      : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of   (The_Queue   : in Queue;
                           Result      : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty    (The_Queue   : in Queue;
                           Result      : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of    (The_Queue   : in Queue;
                           Result      : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

end
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

    OPERATOR Is_Equal
    SPECIFICATION
      INPUT
        Left : Queue,
        Right : Queue
      OUTPUT
        Result : Boolean
      EXCEPTIONS
        Overflow, Underflow
    END

    OPERATOR Length_Of
    SPECIFICATION
      INPUT
        The_Queue : Queue
      OUTPUT
        Result : Natural
      EXCEPTIONS
        Overflow, Underflow
    END

    OPERATOR Is_Empty
    SPECIFICATION
      INPUT
        The_Queue : Queue
      OUTPUT
        Result : Boolean
      EXCEPTIONS
        Overflow, Underflow
    END

    OPERATOR Front_Of
    SPECIFICATION
      INPUT
        The_Queue : Queue
      OUTPUT
        Result : Item
      EXCEPTIONS
        Overflow, Underflow
    END

  END
  IMPLEMENTATION ADA
  Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
  END
```

QUEUE PRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)     return
Priority;
    with function "<="        (Left     : in Priority;
                               Right    : in Priority) return Boolean;
package Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator is

    type Queue(The_Size : Positive) is limited private;

    procedure Copy        (From_The_Queue  : in     Queue;
                           To_The_Queue    : in out Queue);
    procedure Clear       (The_Queue       : in out Queue);
    procedure Add         (The_Item        : in     Item;
                           To_The_Queue    : in out Queue);
    procedure Pop         (The_Queue       : in out Queue);
    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal    (Left           : in Queue;
                           Right          : in Queue;
                           Result         : out Boolean);
    procedure Length_Of   (The_Queue      : in Queue;
                           Result         : out Natural);
    procedure Is_Empty    (The_Queue      : in Queue;
                           Result         : out Boolean);
    procedure Front_Of    (The_Queue      : in Queue;
                           Result         : Item);
    procedure Position_Of (The_Item     : in Item;
                           In_The_Queue : in Queue;
                           Result       : out Natural);

--  end of modification

    function Is_Equal    (Left         : in Queue;
                          Right        : in Queue) return Boolean;
    function Length_Of   (The_Queue    : in Queue) return Natural;
    function Is_Empty    (The_Queue    : in Queue) return Boolean;
    function Front_Of    (The_Queue    : in Queue) return Item;
    function Position_Of (The_Item     : in Item;
                          In_The_Queue : in Queue) return Natural;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Items is array(Positive range <>) of Item;
    type Queue(The_Size : Positive) is
        record
            The_Back  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator is

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
    begin
        if From_The_Queue.The_Back > To_The_Queue.The_Size then
            raise Overflow;
        elsif From_The_Queue.The_Back = 0 then
            To_The_Queue.The_Back := 0;
        else
            To_The_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
              From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
            To_The_Queue.The_Back := From_The_Queue.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue.The_Back := 0;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Index : Natural := 1;
    begin
        if To_The_Queue.The_Back = 0 then
            To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
        else
            while (Index <= To_The_Queue.The_Back) and then
                  (Priority_Of(The_Item) <=
                    Priority_Of(To_The_Queue.The_Items(Index))) loop
                Index := Index + 1;
            end loop;
            if Index > To_The_Queue.The_Back then
                To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
The_Item;
                To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
            else
                To_The_Queue.The_Items
                  ((Index + 1) .. (To_The_Queue.The_Back + 1)) :=
                  To_The_Queue.The_Items(Index ..
To_The_Queue.The_Back);
                To_The_Queue.The_Items(Index) := The_Item;
                To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
            end if;
        end if;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        elsif The_Queue.The_Back = 1 then
            The_Queue.The_Back := 0;
        else
            The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
              The_Queue.The_Items(2 .. The_Queue.The_Back);
            The_Queue.The_Back := The_Queue.The_Back - 1;
        end if;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
    begin
        if From_The_Queue.The_Back < At_The_Position then
            raise Position_Error;
        elsif From_The_Queue.The_Back /= At_The_Position then
            From_The_Queue.The_Items
              (At_The_Position .. (From_The_Queue.The_Back - 1)) :=
                From_The_Queue.The_Items
                  ((At_The_Position + 1) .. From_The_Queue.The_Back);
        end if;
        From_The_Queue.The_Back := From_The_Queue.The_Back - 1;
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left           : in Queue;
                            Right          : in Queue;
                            Result         : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue      : in Queue;
                            Result         : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue      : in Queue;
                            Result         : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue      : in Queue;
                            Result         : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of (The_Item       : in Item;
                           In_The_Queue   : in Queue;
                           Result         : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
    begin
        if Left.The_Back /= Right.The_Back then
            return False;
        else
            for Index in 1 .. Left.The_Back loop
                if Left.The_Items(Index) /= Right.The_Items(Index)
then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
    begin
        return The_Queue.The_Back;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Back = 0);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        else
            return The_Queue.The_Items(1);
        end if;
    end Front_Of;

    function Position_Of (The_Item      : in Item;
                          In_The_Queue  : in Queue) return Natural is
    begin
        for Index in 1 .. In_The_Queue.The_Back loop
            if In_The_Queue.The_Items(Index) = The_Item then
                return Index;
            end if;
        end loop;
        return 0;
    end Position_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        Continue : Boolean;
    begin
        for The_Iterator in 1 .. Over_The_Queue.The_Back loop
            Process(Over_The_Queue.The_Items(The_Iterator), Continue);
            exit when not Continue;
        end loop;
    end Iterate;

end Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Bounded_Managed_Iterator
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)      return
Priority;
    with function "<="          (Left     : in Priority;
                                 Right    : in Priority) return Boolean;
package
Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator is

    type Queue is limited private;

    procedure Copy          (From_The_Queue  : in     Queue;
                             To_The_Queue    : in out Queue);
    procedure Clear         (The_Queue       : in out Queue);
    procedure Add           (The_Item        : in     Item;
                             To_The_Queue    : in out Queue);
    procedure Pop           (The_Queue       : in out Queue);
    procedure Remove_Item (From_The_Queue  : in out Queue;
                             At_The_Position : in     Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left            : in Queue;
                             Right           : in Queue;
                             Result          : out Boolean);
    procedure Length_Of     (The_Queue       : in Queue;
                             Result          : out Natural);
    procedure Is_Empty      (The_Queue       : in Queue;
                             Result          : out Boolean);
    procedure Front_Of      (The_Queue       : in Queue;
                             Result          : Item);
    procedure Position_Of (The_Item        : in Item;
                             In_The_Queue    : in Queue;
                             Result          : out Natural);

--  end of modification

    function Is_Equal      (Left          : in Queue;
                            Right         : in Queue) return Boolean;
    function Length_Of     (The_Queue     : in Queue) return Natural;
    function Is_Empty      (The_Queue     : in Queue) return Boolean;
    function Front_Of      (The_Queue     : in Queue) return Item;
    function Position_Of (The_Item      : in Item;
                            In_The_Queue  : in Queue) return Natural;

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--              "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Structure,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        Node_Manager.Free(To_The_Queue.The_Front);
        To_The_Queue.The_Back := null;
        if From_The_Queue.The_Front /= null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_The_Queue.The_Front.The_Item := From_Index.The_Item;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.The_Item := From_Index.The_Item;
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        Node_Manager.Free(The_Queue.The_Front);
        The_Queue.The_Back := null;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Previous : Structure;
        Index    : Structure := To_The_Queue.The_Front;
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Front.The_Item := The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            while (Index /= null) and then
                    (Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
                Previous := Index;
                Index := Index.Next;
            end loop;
            if Previous = null then
                To_The_Queue.The_Front := Node_Manager.New_Item;
                To_The_Queue.The_Front.The_Item := The_Item;
                To_The_Queue.The_Front.Next := Index;
                if To_The_Queue.The_Back = null then
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                end if;
            elsif Index = null then
                To_The_Queue.The_Back.Next := Node_Manager.New_Item;
                To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
                To_The_Queue.The_Back.The_Item := The_Item;
            else
                Previous.Next := Node_Manager.New_Item;
                Previous.Next.The_Item := The_Item;
                Previous.Next.Next := Index;
            end if;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Queue.The_Front;
        The_Queue.The_Front := The_Queue.The_Front.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
        Index.Next := null;
        Node_Manager.Free(Index);
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left         : in Queue;
                            Right        : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue    : in Queue;
                            Result       : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue    : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue    : in Queue;
                            Result       : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of  (The_Item     : in Item;
                            In_The_Queue : in Queue;
                            Result       : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;
```

```ada
-- end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

    function Position_Of (The_Item     : in Item;
                          In_The_Queue : in Queue) return Natural is
        Position : Natural   := 1;
        Index    : Structure := In_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                return Position;
            else
                Position := Position + 1;
                Index := Index.Next;
            end if;
        end loop;
        return 0;
    end Position_Of;

end Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Unbounded_Managed_Noniterator
END
```

QUEUE PRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)      return
Priority;
    with function "<="          (Left     : in Priority;
                                 Right    : in Priority) return Boolean;
package Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator
is

    type Queue(The_Size : Positive) is limited private;

    procedure Copy  (From_The_Queue : in      Queue;
                     To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in      Item;
                     To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean);
    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural);
    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean);
    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item);

--  end of modification

    function Is_Equal  (Left      : in Queue;
                        Right     : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty  (The_Queue : in Queue) return Boolean;
    function Front_Of  (The_Queue : in Queue) return Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow  : exception;
    Underflow : exception;

private
    type Items is array(Positive range <>) of Item;
    type Queue(The_Size : Positive) is
        record
            The_Back  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator is

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
    begin
        if From_The_Queue.The_Back > To_The_Queue.The_Size then
            raise Overflow;
        elsif From_The_Queue.The_Back = 0 then
            To_The_Queue.The_Back := 0;
        else
            To_The_Queue.The_Items(1 .. From_The_Queue.The_Back) :=
              From_The_Queue.The_Items(1 .. From_The_Queue.The_Back);
            To_The_Queue.The_Back := From_The_Queue.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue.The_Back := 0;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Index : Natural := 1;
    begin
        if To_The_Queue.The_Back = 0 then
            To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
        else
            while (Index <= To_The_Queue.The_Back) and then
                  (Priority_Of(The_Item) <=
                    Priority_Of(To_The_Queue.The_Items(Index))) loop
                Index := Index + 1;
            end loop;
            if Index > To_The_Queue.The_Back then
                To_The_Queue.The_Items(To_The_Queue.The_Back + 1) :=
The_Item;
                To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
            else
                To_The_Queue.The_Items
                  ((Index + 1) .. (To_The_Queue.The_Back + 1)) :=
                    To_The_Queue.The_Items(Index ..
To_The_Queue.The_Back);
                To_The_Queue.The_Items(Index) := The_Item;
                To_The_Queue.The_Back := To_The_Queue.The_Back + 1;
            end if;
        end if;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        elsif The_Queue.The_Back = 1 then
            The_Queue.The_Back := 0;
        else
            The_Queue.The_Items(1 .. (The_Queue.The_Back - 1)) :=
              The_Queue.The_Items(2 .. The_Queue.The_Back);
            The_Queue.The_Back := The_Queue.The_Back - 1;
        end if;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left         : in Queue;
                            Right        : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue    : in Queue;
                            Result       : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue    : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue    : in Queue;
                            Result       : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
    begin
        if Left.The_Back /= Right.The_Back then
            return False;
        else
            for Index in 1 .. Left.The_Back loop
                if Left.The_Items(Index) /= Right.The_Items(Index)
then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
    begin
        return The_Queue.The_Back;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Back = 0);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        if The_Queue.The_Back = 0 then
            raise Underflow;
        else
            return The_Queue.The_Items(1);
        end if;
    end Front_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        Continue : Boolean;
    begin
        for The_Iterator in 1 .. Over_The_Queue.The_Back loop
            Process(Over_The_Queue.The_Items(The_Iterator), Continue);
            exit when not Continue;
        end loop;
    end Iterate;

end Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Bounded_Managed_Iterator
END
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)     return
Priority;
    with function "<="         (Left     : in Priority;
                                Right    : in Priority) return Boolean;
package
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator is

    type Queue is limited private;

    procedure Copy  (From_The_Queue : in      Queue;
                      To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in      Item;
                      To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                             Right         : in Queue;
                             Result        : out Boolean);
    procedure Length_Of    (The_Queue     : in Queue;
                             Result        : out Natural);
    procedure Is_Empty     (The_Queue     : in Queue;
                             Result        : out Boolean);
    procedure Front_Of     (The_Queue     : in Queue;
                             Result        : Item);

--  end of modification

    function Is_Equal  (Left      : in Queue;
                        Right     : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty  (The_Queue : in Queue) return Boolean;
    function Front_Of  (The_Queue : in Queue) return Item;

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
  is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Structure,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        Node_Manager.Free(To_The_Queue.The_Front);
        To_The_Queue.The_Back := null;
        if From_The_Queue.The_Front /= null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_The_Queue.The_Front.The_Item := From_Index.The_Item;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.The_Item := From_Index.The_Item;
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        Node_Manager.Free(The_Queue.The_Front);
        The_Queue.The_Back := null;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Previous : Structure;
        Index    : Structure := To_The_Queue.The_Front;
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Front.The_Item := The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            while (Index /= null) and then
                  (Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
                Previous := Index;
                Index := Index.Next;
            end loop;
            if Previous = null then
                To_The_Queue.The_Front := Node_Manager.New_Item;
                To_The_Queue.The_Front.The_Item := The_Item;
                To_The_Queue.The_Front.Next := Index;
                if To_The_Queue.The_Back = null then
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                end if;
            elsif Index = null then
                To_The_Queue.The_Back.Next := Node_Manager.New_Item;
                To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
                To_The_Queue.The_Back.The_Item := The_Item;
            else
                Previous.Next := Node_Manager.New_Item;
                Previous.Next.The_Item := The_Item;
                Previous.Next.Next := Index;
            end if;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Queue.The_Front;
        The_Queue.The_Front := The_Queue.The_Front.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left          : in Queue;
                             Right         : in Queue;
                             Result        : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_Queue     : in Queue;
                             Result        : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty      (The_Queue     : in Queue;
                             Result        : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of      (The_Queue     : in Queue;
                             Result        : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
```

```
end Is_Empty;

function Front_Of (The_Queue : in Queue) return Item is
begin
    return The_Queue.The_Front.The_Item;
exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

end
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Unbounded_Managed_Noniterator
END
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator is

    type Queue is limited private;

    procedure Copy          (From_The_Queue   : in      Queue;
                             To_The_Queue     : in out Queue);
    procedure Clear         (The_Queue        : in out Queue);
    procedure Add           (The_Item         : in      Item;
                             To_The_Queue     : in out Queue);
    procedure Pop           (The_Queue        : in out Queue);
    procedure Remove_Item   (From_The_Queue   : in out Queue;
                             At_The_Position  : in      Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left          : in Queue;
                             Right         : in Queue;
                             Result        : out Boolean);
    procedure Length_Of     (The_Queue     : in Queue;
                             Result        : out Natural);
    procedure Is_Empty      (The_Queue     : in Queue;
                             Result        : out Boolean);
    procedure Front_Of      (The_Queue     : in Queue;
                             Result        : Item);
    procedure Position_Of   (The_Item      : in Item;
                             In_The_Queue  : in Queue;
                             Result        : out Natural);

--  end of modification

    function Is_Equal    (Left          : in Queue;
                          Right         : in Queue) return Boolean;
    function Length_Of   (The_Queue     : in Queue) return Natural;
    function Is_Empty    (The_Queue     : in Queue) return Boolean;
    function Front_Of    (The_Queue     : in Queue) return Item;
    function Position_Of (The_Item      : in Item;
                          In_The_Queue  : in Queue) return Natural;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
  is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                    Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of (The_Item     : in Item;
                           In_The_Queue : in Queue;
                           Result       : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

    function Position_Of (The_Item     : in Item;
                          In_The_Queue : in Queue) return Natural is
        Position : Natural   := 1;
```

```
    Index     : Structure := In_The_Queue.The_Front;
begin
    while Index /= null loop
        if Index.The_Item = The_Item then
            return Position;
        else
            Position := Position + 1;
            Index := Index.Next;
        end if;
    end loop;
    return 0;
end Position_Of;
```

```
    procedure Iterate (Over_The_Queue : in Queue) is
        The_Iterator : Structure := Over_The_Queue.The_Front;
        Continue     : Boolean;
    begin
        while not (The_Iterator = null) loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Iterator
END
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
is

    type Queue is limited private;

    procedure Copy  (From_The_Queue : in     Queue;
                     To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in     Item;
                     To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean);
    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural);
    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean);
    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item);

--  end of modification

    function Is_Equal  (Left       : in Queue;
                        Right      : in Queue) return Boolean;
    function Length_Of (The_Queue  : in Queue) return Natural;
    function Is_Empty  (The_Queue  : in Queue) return Boolean;
    function Front_Of  (The_Queue  : in Queue) return Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
  Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                    Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left           : in Queue;
                             Right          : in Queue;
                             Result         : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_Queue      : in Queue;
                             Result         : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty      (The_Queue      : in Queue;
                             Result         : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of      (The_Queue      : in Queue;
                             Result         : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        The_Iterator : Structure := Over_The_Queue.The_Front;
        Continue     : Boolean;
    begin
        while not (The_Iterator = null) loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)     return
Priority;
    with function "<="         (Left     : in Priority;
                                Right    : in Priority) return Boolean;
package Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator
is

    type Queue is limited private;

    procedure Copy        (From_The_Queue  : in     Queue;
                           To_The_Queue    : in out Queue);
    procedure Clear       (The_Queue       : in out Queue);
    procedure Add         (The_Item        : in     Item;
                           To_The_Queue    : in out Queue);
    procedure Pop         (The_Queue       : in out Queue);
    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive);

    function Is_Equal     (Left            : in Queue;
                           Right           : in Queue) return Boolean;
    function Length_Of    (The_Queue       : in Queue) return Natural;
    function Is_Empty     (The_Queue       : in Queue) return Boolean;
    function Front_Of     (The_Queue       : in Queue) return Item;
    function Position_Of  (The_Item        : in Item;
                           In_The_Queue    : in Queue) return Natural;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Previous : Structure;
        Index    : Structure := To_The_Queue.The_Front;
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            while (Index /= null) and then
                  (Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
                Previous := Index;
                Index := Index.Next;
            end loop;
            if Previous = null then
                To_The_Queue.The_Front :=
                  new Node'(The_Item => The_Item,
                            Next     => Index);
                if To_The_Queue.The_Back = null then
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                end if;
            elsif Index = null then
                To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                        Next     =>
null);
                To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
            else
                Previous.Next := new Node'(The_Item => The_Item,
                                           Next     => Index);
            end if;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left         : in Queue;
                            Right        : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue    : in Queue;
                            Result       : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue    : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue    : in Queue;
                            Result       : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of (The_Item      : in Item;
                            In_The_Queue : in Queue;
                            Result       : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
```

```
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Length_Of;

function Is_Empty (The_Queue : in Queue) return Boolean is
begin
    return (The_Queue.The_Front = null);
end Is_Empty;

function Front_Of (The_Queue : in Queue) return Item is
begin
    return The_Queue.The_Front.The_Item;
exception
    when Constraint_Error =>
        raise Underflow;
end Front_Of;

function Position_Of (The_Item     : in Item;
                      In_The_Queue : in Queue) return Natural is
    Position : Natural   := 1;
    Index    : Structure := In_The_Queue.The_Front;
begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                return Position;
            else
                Position := Position + 1;
                Index := Index.Next;
            end if;
        end loop;
        return 0;
    end Position_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        The_Iterator : Structure := Over_The_Queue.The_Front;
        Continue     : Boolean;
    begin
        while not (The_Iterator = null) loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Iterator
END
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)     return
Priority;
    with function "<="        (Left     : in Priority;
                               Right    : in Priority) return Boolean;
package
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator is

    type Queue is limited private;

    procedure Copy  (From_The_Queue : in     Queue;
                     To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in     Item;
                     To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean);
    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural);
    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean);
    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item);

--  end of modification

    function Is_Equal  (Left      : in Queue;
                        Right     : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty  (The_Queue : in Queue) return Boolean;
    function Front_Of  (The_Queue : in Queue) return Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION


```
package body
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
  is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Previous : Structure;
        Index    : Structure := To_The_Queue.The_Front;
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            while (Index /= null) and then
                  (Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
                Previous := Index;
                Index := Index.Next;
            end loop;
            if Previous = null then
                To_The_Queue.The_Front :=
                  new Node'(The_Item => The_Item,
                            Next     => Index);
                if To_The_Queue.The_Back = null then
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                end if;
            elsif Index = null then
                To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                        Next     =>
null);
                To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
            else
                Previous.Next := new Node'(The_Item => The_Item,
                                           Next     => Index);
            end if;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left         : in Queue;
                             Right        : in Queue;
                             Result       : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_Queue    : in Queue;
                             Result       : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty      (The_Queue    : in Queue;
                             Result       : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of      (The_Queue    : in Queue;
                             Result       : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        The_Iterator : Structure := Over_The_Queue.The_Front;
        Continue     : Boolean;
    begin
        while not (The_Iterator = null) loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Iterator
END
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator is

    type Queue is limited private;

    procedure Copy          (From_The_Queue  : in      Queue;
                             To_The_Queue    : in out Queue);
    procedure Clear         (The_Queue       : in out Queue);
    procedure Add           (The_Item        : in      Item;
                             To_The_Queue    : in out Queue);
    procedure Pop           (The_Queue       : in out Queue);
    procedure Remove_Item   (From_The_Queue  : in out Queue;
                             At_The_Position : in      Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left          : in Queue;
                             Right         : in Queue;
                             Result        : out Boolean);
    procedure Length_Of     (The_Queue     : in Queue;
                             Result        : out Natural);
    procedure Is_Empty      (The_Queue     : in Queue;
                             Result        : out Boolean);
    procedure Front_Of      (The_Queue     : in Queue;
                             Result        : Item);
    procedure Position_Of   (The_Item      : in Item;
                             In_The_Queue  : in Queue;
                             Result        : out Natural);

--  end of modification

    function Is_Equal     (Left          : in Queue;
                           Right         : in Queue) return Boolean;
    function Length_Of    (The_Queue     : in Queue) return Natural;
    function Is_Empty     (The_Queue     : in Queue) return Boolean;
    function Front_Of     (The_Queue     : in Queue) return Item;
    function Position_Of  (The_Item      : in Item;
                           In_The_Queue  : in Queue) return Natural;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION


```
with Storage_Manager_Sequential;
package body
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                  (Item        => Node,
                                   Pointer     => Structure,
                                   Free        => Free,
                                   Set_Pointer => Set_Next,
                                   Pointer_Of  => Next_Of);

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        Node_Manager.Free(To_The_Queue.The_Front);
        To_The_Queue.The_Back := null;
        if From_The_Queue.The_Front /= null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_The_Queue.The_Front.The_Item := From_Index.The_Item;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.The_Item := From_Index.The_Item;
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        Node_Manager.Free(The_Queue.The_Front);
        The_Queue.The_Back := null;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Front.The_Item := The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            To_The_Queue.The_Back.Next := Node_Manager.New_Item;
            To_The_Queue.The_Back.Next.The_Item:= The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Queue.The_Front;
        The_Queue.The_Front := The_Queue.The_Front.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
        Index.Next := null;
        Node_Manager.Free(Index);
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left          : in Queue;
                             Right         : in Queue;
                             Result        : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_Queue     : in Queue;
                             Result        : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty      (The_Queue     : in Queue;
                             Result        : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of      (The_Queue     : in Queue;
                             Result        : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of (The_Item     : in Item;
                           In_The_Queue : in Queue;
                           Result       : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;
```

```
function Length_Of (The_Queue : in Queue) return Natural is
    Count : Natural   := 0;
    Index : Structure := The_Queue.The_Front;
begin
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Length_Of;

function Is_Empty (The_Queue : in Queue) return Boolean is
begin
    return (The_Queue.The_Front = null);
end Is_Empty;

function Front_Of (The_Queue : in Queue) return Item is
begin
    return The_Queue.The_Front.The_Item;
exception
    when Constraint_Error =>
        raise Underflow;
end Front_Of;

function Position_Of (The_Item     : in Item;
                      In_The_Queue : in Queue) return Natural is
    Position : Natural   := 1;
    Index    : Structure := In_The_Queue.The_Front;
begin
    while Index /= null loop
        if Index.The_Item = The_Item then
            return Position;
        else
            Position := Position + 1;
            Index := Index.Next;
        end if;
    end loop;
    return 0;
end Position_Of;

procedure Iterate (Over_The_Queue : in Queue) is
    The_Iterator : Structure := Over_The_Queue.The_Front;
    Continue     : Boolean;
begin
    while not (The_Iterator = null) loop
        Process(The_Iterator.The_Item, Continue);
        exit when not Continue;
        The_Iterator := The_Iterator.Next;
    end loop;
end Iterate;

end Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Balking_Sequential_Unbounded_Managed_Iterator
END
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
is

    type Queue is limited private;

    procedure Copy        (From_The_Queue  : in      Queue;
                           To_The_Queue    : in out Queue);
    procedure Clear       (The_Queue       : in out Queue);
    procedure Add         (The_Item        : in      Item;
                           To_The_Queue    : in out Queue);
    procedure Pop         (The_Queue       : in out Queue);
    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in      Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal    (Left            : in Queue;
                           Right           : in Queue;
                           Result          : out Boolean);
    procedure Length_Of   (The_Queue       : in Queue;
                           Result          : out Natural);
    procedure Is_Empty    (The_Queue       : in Queue;
                           Result          : out Boolean);
    procedure Front_Of    (The_Queue       : in Queue;
                           Result          : Item);
    procedure Position_Of (The_Item     : in Item;
                           In_The_Queue : in Queue;
                           Result       : out Natural);

--  end of modification

    function Is_Equal    (Left         : in Queue;
                          Right        : in Queue) return Boolean;
    function Length_Of   (The_Queue    : in Queue) return Natural;
    function Is_Empty    (The_Queue    : in Queue) return Boolean;
    function Front_Of    (The_Queue    : in Queue) return Item;
    function Position_Of (The_Item     : in Item;
                          In_The_Queue : in Queue) return Natural;

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
  Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                    Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal    (Left          : in Queue;
                           Right         : in Queue;
                           Result        : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of   (The_Queue     : in Queue;
                           Result        : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty    (The_Queue     : in Queue;
                           Result        : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of    (The_Queue     : in Queue;
                           Result        : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of (The_Item      : in Item;
                           In_The_Queue  : in Queue;
                           Result        : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;
```

```
function Position_Of (The_Item     : in Item;
                      In_The_Queue : in Queue) return Natural is
    Position : Natural   := 1;
    Index    : Structure := In_The_Queue.The_Front;
begin
    while Index /= null loop
        if Index.The_Item = The_Item then
            return Position;
        else
            Position := Position + 1;
            Index := Index.Next;
        end if;
    end loop;
    return 0;
end Position_Of;

end
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE NONPRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
END
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
r
  is

    type Queue is limited private;

    procedure Copy  (From_The_Queue : in     Queue;
                     To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in     Item;
                     To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean);
    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural);
    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean);
    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item);

--  end of modification

    function Is_Equal  (Left      : in Queue;
                        Right     : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty  (The_Queue : in Queue) return Boolean;
    function Front_Of  (The_Queue : in Queue) return Item;

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
r;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body

Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
r is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                    Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left         : in Queue;
                            Right        : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue    : in Queue;
                            Result       : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue    : in Queue;
                            Result       : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue    : in Queue;
                            Result       : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

end
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
r;
```

QUEUE NONPRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
r
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterato
r
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)      return
Priority;
    with function "<="         (Left     : in Priority;
                                Right    : in Priority) return Boolean;
package Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
is

    type Queue is limited private;

    procedure Copy          (From_The_Queue  : in     Queue;
                             To_The_Queue    : in out Queue);
    procedure Clear         (The_Queue       : in out Queue);
    procedure Add           (The_Item        : in     Item;
                             To_The_Queue    : in out Queue);
    procedure Pop           (The_Queue       : in out Queue);
    procedure Remove_Item   (From_The_Queue  : in out Queue;
                             At_The_Position : in     Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left            : in Queue;
                             Right           : in Queue;
                             Result          : out Boolean);
    procedure Length_Of     (The_Queue       : in Queue;
                             Result          : out Natural);
    procedure Is_Empty      (The_Queue       : in Queue;
                             Result          : out Boolean);
    procedure Front_Of      (The_Queue       : in Queue;
                             Result          : Item);
    procedure Position_Of (The_Item     : in Item;
                           In_The_Queue : in Queue;
                           Result       : out Natural);

--  end of modification

    function Is_Equal     (Left         : in Queue;
                           Right        : in Queue) return Boolean;
    function Length_Of    (The_Queue    : in Queue) return Natural;
    function Is_Empty     (The_Queue    : in Queue) return Boolean;
    function Front_Of     (The_Queue    : in Queue) return Item;
    function Position_Of  (The_Item     : in Item;
                           In_The_Queue : in Queue) return Natural;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Queue : in Queue);

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Structure,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        Node_Manager.Free(To_The_Queue.The_Front);
        To_The_Queue.The_Back := null;
        if From_The_Queue.The_Front /= null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_The_Queue.The_Front.The_Item := From_Index.The_Item;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.The_Item := From_Index.The_Item;
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        Node_Manager.Free(The_Queue.The_Front);
        The_Queue.The_Back := null;
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Previous : Structure;
        Index    : Structure := To_The_Queue.The_Front;
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := Node_Manager.New_Item;
            To_The_Queue.The_Front.The_Item := The_Item;
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            while (Index /= null) and then
                  (Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
                Previous := Index;
                Index := Index.Next;
            end loop;
            if Previous = null then
                To_The_Queue.The_Front := Node_Manager.New_Item;
                To_The_Queue.The_Front.The_Item := The_Item;
                To_The_Queue.The_Front.Next := Index;
                if To_The_Queue.The_Back = null then
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                end if;
            elsif Index = null then
                To_The_Queue.The_Back.Next := Node_Manager.New_Item;
                To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
                To_The_Queue.The_Back.The_Item := The_Item;
            else
                Previous.Next := Node_Manager.New_Item;
                Previous.Next.The_Item := The_Item;
                Previous.Next.Next := Index;
            end if;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Queue.The_Front;
        The_Queue.The_Front := The_Queue.The_Front.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
        Index.Next := null;
        Node_Manager.Free(Index);
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left           : in Queue;
                            Right          : in Queue;
                            Result         : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue      : in Queue;
                            Result         : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue      : in Queue;
                            Result         : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue      : in Queue;
                            Result         : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of  (The_Item       : in Item;
                            In_The_Queue   : in Queue;
                            Result         : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;
```

```
-- end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

    function Position_Of (The_Item     : in Item;
                          In_The_Queue : in Queue) return Natural is
        Position : Natural   := 1;
        Index    : Structure := In_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                return Position;
            else
                Position := Position + 1;
                Index := Index.Next;
            end if;
        end loop;
        return 0;
    end Position_Of;

    procedure Iterate (Over_The_Queue : in Queue) is
        The_Iterator : Structure := Over_The_Queue.The_Front;
        Continue     : Boolean;
    begin
        while not (The_Iterator = null) loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Unbounded_Managed_Iterator
END
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)      return
Priority;
    with function "<="         (Left      : in Priority;
                                Right     : in Priority) return Boolean;
package
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator is

    type Queue is limited private;

    procedure Copy         (From_The_Queue  : in      Queue;
                            To_The_Queue    : in out Queue);
    procedure Clear        (The_Queue       : in out Queue);
    procedure Add          (The_Item        : in      Item;
                            To_The_Queue    : in out Queue);
    procedure Pop          (The_Queue       : in out Queue);
    procedure Remove_Item  (From_The_Queue  : in out Queue;
                            At_The_Position : in      Positive);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left            : in Queue;
                            Right           : in Queue;
                            Result          : out Boolean);
    procedure Length_Of    (The_Queue       : in Queue;
                            Result          : out Natural);
    procedure Is_Empty     (The_Queue       : in Queue;
                            Result          : out Boolean);
    procedure Front_Of     (The_Queue       : in Queue;
                            Result          : Item);
    procedure Position_Of  (The_Item        : in Item;
                            In_The_Queue    : in Queue;
                            Result          : out Natural);

--  end of modification

    function Is_Equal      (Left            : in Queue;
                            Right           : in Queue) return Boolean;
    function Length_Of     (The_Queue       : in Queue) return Natural;
    function Is_Empty      (The_Queue       : in Queue) return Boolean;
    function Front_Of      (The_Queue       : in Queue) return Item;
    function Position_Of   (The_Item        : in Item;
                            In_The_Queue    : in Queue) return Natural;

    Overflow       : exception;
    Underflow      : exception;
    Position_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--           "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
  is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in     Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Previous : Structure;
        Index    : Structure := To_The_Queue.The_Front;
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            while (Index /= null) and then
                  (Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
                Previous := Index;
                Index := Index.Next;
            end loop;
            if Previous = null then
                To_The_Queue.The_Front :=
                  new Node'(The_Item => The_Item,
                            Next     => Index);
                if To_The_Queue.The_Back = null then
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                end if;
            elsif Index = null then
                To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                        Next     =>
null);
                To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
            else
                Previous.Next := new Node'(The_Item => The_Item,
                                           Next     => Index);
            end if;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Remove_Item (From_The_Queue  : in out Queue;
                           At_The_Position : in     Positive) is
        Count    : Natural   := 1;
        Previous : Structure;
        Index    : Structure := From_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Count = At_The_Position then
                exit;
            else
                Count := Count + 1;
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        if Index = null then
            raise Position_Error;
        elsif Previous = null then
            From_The_Queue.The_Front := Index.Next;
        else
            Previous.Next := Index.Next;
        end if;
        if From_The_Queue.The_Back = Index then
            From_The_Queue.The_Back := Previous;
        end if;
    end Remove_Item;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left          : in Queue;
                            Right         : in Queue;
                            Result        : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of    (The_Queue     : in Queue;
                            Result        : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty     (The_Queue     : in Queue;
                            Result        : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of     (The_Queue     : in Queue;
                            Result        : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

    procedure Position_Of (The_Item      : in Item;
                           In_The_Queue  : in Queue;
                           Result        : out Natural) is
    begin
        Result := Position_Of(The_Item,In_The_Queue);
    end Position_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural   := 0;
        Index : Structure := The_Queue.The_Front;
    begin
```

```
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Length_Of;

function Is_Empty (The_Queue : in Queue) return Boolean is
begin
    return (The_Queue.The_Front = null);
end Is_Empty;

function Front_Of (The_Queue : in Queue) return Item is
begin
    return The_Queue.The_Front.The_Item;
exception
    when Constraint_Error =>
        raise Underflow;
end Front_Of;
```

```
    function Position_Of (The_Item     : in Item;
                          In_The_Queue : in Queue) return Natural is
        Position : Natural   := 1;
        Index    : Structure := In_The_Queue.The_Front;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                return Position;
            else
                Position := Position + 1;
                Index := Index.Next;
            end if;
        end loop;
        return 0;
    end Position_Of;

end Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE PRIORITY BALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Remove_Item
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      At_The_Position : Positive
    OUTPUT
      From_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

  OPERATOR Position_Of
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Position_Error
  END

END
IMPLEMENTATION ADA
Queue_Priority_Balking_Sequential_Unbounded_Unmanaged_Noniterator
END
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Priority is limited private;
    with function Priority_Of (The_Item : in Item)     return
Priority;
    with function "<="       (Left     : in Priority;
                              Right    : in Priority) return Boolean;
package
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
is

    type Queue is limited private;

    procedure Copy  (From_The_Queue : in     Queue;
                     To_The_Queue   : in out Queue);
    procedure Clear (The_Queue      : in out Queue);
    procedure Add   (The_Item       : in     Item;
                     To_The_Queue   : in out Queue);
    procedure Pop   (The_Queue      : in out Queue);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal     (Left         : in Queue;
                            Right        : in Queue;
                            Result       : out Boolean);
    procedure Length_Of    (The_Queue    : in Queue;
                            Result       : out Natural);
    procedure Is_Empty     (The_Queue    : in Queue;
                            Result       : out Boolean);
    procedure Front_Of     (The_Queue    : in Queue;
                            Result       : Item);

--  end of modification

    function Is_Equal  (Left      : in Queue;
                        Right     : in Queue) return Boolean;
    function Length_Of (The_Queue : in Queue) return Natural;
    function Is_Empty  (The_Queue : in Queue) return Boolean;
    function Front_Of  (The_Queue : in Queue) return Item;

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Structure is access Node;
    type Queue is
        record
            The_Front : Structure;
            The_Back  : Structure;
        end record;
end
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
  Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
is

    type Node is
        record
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Queue : in      Queue;
                    To_The_Queue   : in out Queue) is
        From_Index : Structure := From_The_Queue.The_Front;
        To_Index   : Structure;
    begin
        if From_The_Queue.The_Front = null then
            To_The_Queue.The_Front := null;
            To_The_Queue.The_Back := null;
        else
            To_The_Queue.The_Front :=
              new Node'(The_Item => From_Index.The_Item,
                        Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
            To_Index := To_The_Queue.The_Front;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
                To_The_Queue.The_Back := To_Index;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Queue : in out Queue) is
    begin
        The_Queue := Queue'(The_Front => null,
                            The_Back  => null);
    end Clear;

    procedure Add (The_Item     : in     Item;
                   To_The_Queue : in out Queue) is
        Previous : Structure;
        Index    : Structure := To_The_Queue.The_Front;
    begin
        if To_The_Queue.The_Front = null then
            To_The_Queue.The_Front := new Node'(The_Item => The_Item,
                                                Next     => null);
            To_The_Queue.The_Back := To_The_Queue.The_Front;
        else
            while (Index /= null) and then
              (Priority_Of(The_Item) <=
Priority_Of(Index.The_Item)) loop
                Previous := Index;
                Index := Index.Next;
            end loop;
            if Previous = null then
                To_The_Queue.The_Front :=
                  new Node'(The_Item => The_Item,
                            Next     => Index);
                if To_The_Queue.The_Back = null then
                    To_The_Queue.The_Back := To_The_Queue.The_Front;
                end if;
            elsif Index = null then
                To_The_Queue.The_Back.Next := new Node'(The_Item =>
The_Item,
                                                        Next     =>
null);
                To_The_Queue.The_Back := To_The_Queue.The_Back.Next;
            else
                Previous.Next := new Node'(The_Item => The_Item,
                                           Next     => Index);
            end if;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Pop (The_Queue : in out Queue) is
    begin
        The_Queue.The_Front := The_Queue.The_Front.Next;
        if The_Queue.The_Front = null then
            The_Queue.The_Back := null;
        end if;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal      (Left          : in Queue;
                             Right         : in Queue;
                             Result        : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Length_Of     (The_Queue     : in Queue;
                             Result        : out Natural) is
    begin
        Result := Length_Of(The_Queue);
    end Length_Of;

    procedure Is_Empty      (The_Queue     : in Queue;
                             Result        : out Boolean) is
    begin
        Result := Is_Empty(The_Queue);
    end Is_Empty;

    procedure Front_Of      (The_Queue     : in Queue;
                             Result        : Item) is
    begin
        Result := Front_Of(The_Queue);
    end Front_Of;

--  end of modification

    function Is_Equal (Left  : in Queue;
                       Right : in Queue) return Boolean is
        Left_Index  : Structure := Left.The_Front;
        Right_Index : Structure := Right.The_Front;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /= Right_Index.The_Item then
                return False;
            else
                Left_Index := Left_Index.Next;
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Length_Of (The_Queue : in Queue) return Natural is
        Count : Natural    := 0;
        Index : Structure := The_Queue.The_Front;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Length_Of;

    function Is_Empty (The_Queue : in Queue) return Boolean is
    begin
        return (The_Queue.The_Front = null);
    end Is_Empty;

    function Front_Of (The_Queue : in Queue) return Item is
    begin
        return The_Queue.The_Front.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Front_Of;

end
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;
```

QUEUE PRIORITY NONBALKING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Priority : PRIVATE_TYPE,
    Priority_Of : FUNCTION[The_Item : Item, RETURN : Priority],
    func_"<=" : FUNCTION[Left : Priority, Right : Priority, RETURN :
Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Queue : Queue,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Queue : Queue
    OUTPUT
      To_The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      The_Queue : Queue
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Queue,
      Right : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Queue : Queue
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Front_Of
  SPECIFICATION
    INPUT
      The_Queue : Queue,
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Queue_Priority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
END
```

RING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Ring_Sequential_Bounded_Managed_Iterator is

    type Ring(The_Size : Positive) is limited private;

    type Direction is (Forward, Backward);

    procedure Copy              (From_The_Ring     : in
Ring;
                                 To_The_Ring       : in out
Ring);
    procedure Clear             (The_Ring          : in out
Ring);
    procedure Insert            (The_Item          : in
Item;
                                 In_The_Ring       : in out
Ring);
    procedure Pop               (The_Ring          : in out
Ring);
    procedure Rotate            (The_Ring          : in out
Ring;
                                 In_The_Direction  : in
Direction);
    procedure Mark              (The_Ring          : in out
Ring);
    procedure Rotate_To_Mark    (The_Ring          : in out
Ring);

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal   (Left      : in Ring;
                          Right     : in Ring;
                          Result    : out Boolean);
    procedure Extent_Of  (The_Ring  : in Ring;
                          Result    : out Natural);
    procedure Is_Empty   (The_Ring  : in Ring;
                          Result    : out Boolean);
    procedure Top_Of     (The_Ring : in Ring;
                          Result   : out Item);
    procedure At_Mark    (The_Ring : in Ring;
                          Result   : out Boolean);

--  end of modification

    function Is_Equal   (Left      : in Ring;
                         Right     : in Ring) return
Boolean;
    function Extent_Of  (The_Ring  : in Ring) return
Natural;
    function Is_Empty   (The_Ring  : in Ring) return
Boolean;
    function Top_Of     (The_Ring  : in Ring) return
Item;
    function At_Mark    (The_Ring  : in Ring) return
Boolean;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out
Boolean);
    procedure Iterate (Over_The_Ring : in Ring);

    Overflow     : exception;
    Underflow    : exception;
    Rotate_Error : exception;

private
    type Items is array(Positive range <>) of Item;
    type Ring(The_Size : Positive) is
        record
            The_Top   : Natural := 0;
            The_Back  : Natural := 0;
            The_Mark  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Ring_Sequential_Bounded_Managed_Iterator;
```

RING SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION


```
package body Ring_Sequential_Bounded_Managed_Iterator
is

    procedure Copy (From_The_Ring : in     Ring;
                    To_The_Ring   : in out Ring) is
    begin
        if From_The_Ring.The_Back >
To_The_Ring.The_Size then
            raise Overflow;
        elsif From_The_Ring.The_Back = 0 then
            To_The_Ring.The_Top := 0;
            To_The_Ring.The_Back := 0;
            To_The_Ring.The_Mark := 0;
        else
            To_The_Ring.The_Items(1 ..
From_The_Ring.The_Back) :=
              From_The_Ring.The_Items(1 ..
From_The_Ring.The_Back);
            To_The_Ring.The_Top :=
From_The_Ring.The_Top;
            To_The_Ring.The_Back :=
From_The_Ring.The_Back;
            To_The_Ring.The_Mark :=
From_The_Ring.The_Mark;
        end if;
    end Copy;

    procedure Clear (The_Ring : in out Ring) is
    begin
        The_Ring.The_Top := 0;
        The_Ring.The_Back := 0;
        The_Ring.The_Mark := 0;
    end Clear;

    procedure Insert (The_Item    : in     Item;
                      In_The_Ring : in out Ring) is
    begin
        if In_The_Ring.The_Back = In_The_Ring.The_Size
then
            raise Overflow;
        elsif In_The_Ring.The_Back = 0 then
            In_The_Ring.The_Top := 1;
            In_The_Ring.The_Back := 1;
            In_The_Ring.The_Mark := 1;
            In_The_Ring.The_Items(1) := The_Item;
        else
            In_The_Ring.The_Items
              ((In_The_Ring.The_Top + 1) ..
(In_The_Ring.The_Back + 1)) :=
              In_The_Ring.The_Items(In_The_Ring.The_Top
..

In_The_Ring.The_Back);
            In_The_Ring.The_Items(In_The_Ring.The_Top)
:= The_Item;
            In_The_Ring.The_Back :=
In_The_Ring.The_Back + 1;
            if In_The_Ring.The_Mark >=
In_The_Ring.The_Top then
                In_The_Ring.The_Mark :=
In_The_Ring.The_Mark + 1;
            end if;
        end if;
    end Insert;

    procedure Pop(The_Ring : in out Ring) is
    begin
        if The_Ring.The_Back = 0 then
            raise Underflow;
        elsif The_Ring.The_Back = 1 then
            The_Ring.The_Top := 0;
            The_Ring.The_Back := 0;
            The_Ring.The_Mark := 0;
        else
            The_Ring.The_Items(The_Ring.The_Top ..
(The_Ring.The_Back - 1)) :=
              The_Ring.The_Items((The_Ring.The_Top + 1)
.. The_Ring.The_Back);
            The_Ring.The_Back := The_Ring.The_Back - 1;
            if The_Ring.The_Top > The_Ring.The_Back
then
                if The_Ring.The_Top = The_Ring.The_Mark
then
                    The_Ring.The_Mark := 1;
                end if;
                The_Ring.The_Top := 1;
            else
                if The_Ring.The_Mark > The_Ring.The_Top
then
                    The_Ring.The_Mark :=
The_Ring.The_Mark - 1;
                end if;
            end if;
        end if;
    end Pop;

    procedure Rotate (The_Ring         : in out Ring;
                      In_The_Direction : in
Direction) is
    begin
        if The_Ring.The_Back = 0 then
            raise Rotate_Error;
        elsif In_The_Direction = Forward then
            The_Ring.The_Top := The_Ring.The_Top + 1;
            if The_Ring.The_Top > The_Ring.The_Back
then
                The_Ring.The_Top := 1;
            end if;
        else
            The_Ring.The_Top := The_Ring.The_Top - 1;
            if The_Ring.The_Top = 0 then
                The_Ring.The_Top := The_Ring.The_Back;
            end if;
        end if;
    end Rotate;

    procedure Mark (The_Ring : in out Ring) is
    begin
        The_Ring.The_Mark := The_Ring.The_Top;
    end Mark;

    procedure Rotate_To_Mark (The_Ring : in out Ring)
is
    begin
        The_Ring.The_Top := The_Ring.The_Mark;
    end Rotate_To_Mark;

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural) is
    begin
        Result := Extent_Of(The_Ring);
    end Extent_Of;

    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Empty(The_Ring);
    end Is_Empty;

    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item) is
    begin
        Result := Top_Of(The_Ring);
    end Top_Of;

    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := At_Mark(The_Ring);
    end At_Mark;

--  end of modification

    function Is_Equal (Left  : in Ring;
                       Right : in Ring) return Boolean
is
        Left_Index  : Natural := Left.The_Top;
        Right_Index : Natural := Right.The_Top;
    begin
        if Left.The_Back /= Right.The_Back then
            return False;
        elsif Left.The_Items(Left_Index) /=
Right.The_Items(Right_Index) then
            return False;
        elsif (Left.The_Mark = Left_Index) and then
            (Right.The_Mark /= Right_Index) then
            return False;
        else
```

```
            Left_Index := Left_Index + 1;
            if Left_Index > Left.The_Back then
                Left_Index := 1;
            end if;
            Right_Index := Right_Index + 1;
            if Right_Index > Right.The_Back then
                Right_Index := 1;
            end if;
            while Left_Index /= Left.The_Top loop
                if Left.The_Items(Left_Index) /=
                   Right.The_Items(Right_Index) then
                    return False;
                elsif (Left.The_Mark = Left_Index) and
then
                    (Right.The_Mark /= Right_Index)
then
                    return False;
                else
                    Left_Index := Left_Index + 1;
                    if Left_Index > Left.The_Back then
                        Left_Index := 1;
                    end if;
                    Right_Index := Right_Index + 1;
                    if Right_Index > Right.The_Back
then
                        Right_Index := 1;
                    end if;
                end if;
            end loop;
            return (Right_Index = Right.The_Top);
        end if;
    exception
        when Constraint_Error =>
            return (Left.The_Top = Right.The_Top);
    end Is_Equal;

    function Extent_Of (The_Ring : in Ring) return
Natural is
    begin
        return The_Ring.The_Back;
    end Extent_Of;

    function Is_Empty (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Back = 0);
    end Is_Empty;

    function Top_Of (The_Ring : in Ring) return Item is
    begin
        return The_Ring.The_Items(The_Ring.The_Top);
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    function At_Mark (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = The_Ring.The_Mark);
    end At_Mark;

    procedure Iterate (Over_The_Ring : in Ring) is
        Continue : Boolean := True;
    begin
        for The_Iterator in Over_The_Ring.The_Top ..
                            Over_The_Ring.The_Back loop

Process(Over_The_Ring.The_Items(The_Iterator),
Continue);
            exit when not Continue;
        end loop;
        if Continue then
            for The_Iterator in 1 ..
Over_The_Ring.The_Top - 1 loop

Process(Over_The_Ring.The_Items(The_Iterator),
Continue);
                exit when not Continue;
            end loop;
        end if;
    end Iterate;

end Ring_Sequential_Bounded_Managed_Iterator;
```

RING SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Ring_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Ring : Ring,
      To_The_Ring : Ring
    OUTPUT
      To_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Ring : Ring
    OUTPUT
      In_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate
  SPECIFICATION
    INPUT
      The_Ring : Ring,
      In_The_Direction : Direction
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate_To_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Ring,
      Right : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR At_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
    INPUT
      Over_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

END
IMPLEMENTATION ADA
Ring_Sequential_Bounded_Managed_Iterator
END
```

RING SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Ring_Sequential_Bounded_Managed_Noniterator is

    type Ring(The_Size : Positive) is limited private;

    type Direction is (Forward, Backward);

    procedure Copy             (From_The_Ring     : in
Ring;
                                To_The_Ring       : in out
Ring);
    procedure Clear            (The_Ring          : in out
Ring);
    procedure Insert           (The_Item          : in
Item;
                                In_The_Ring       : in out
Ring);
    procedure Pop              (The_Ring          : in out
Ring);
    procedure Rotate           (The_Ring          : in out
Ring;
                                In_The_Direction : in
Direction);
    procedure Mark             (The_Ring          : in out
Ring);
    procedure Rotate_To_Mark (The_Ring          : in out
Ring);
```

```
    function Is_Equal  (Left     : in Ring;
                        Right    : in Ring) return
Boolean;
    function Extent_Of (The_Ring : in Ring) return
Natural;
    function Is_Empty  (The_Ring : in Ring) return
Boolean;
    function Top_Of    (The_Ring : in Ring) return
Item;
    function At_Mark   (The_Ring : in Ring) return
Boolean;

    Overflow     : exception;
    Underflow    : exception;
    Rotate_Error : exception;

private
    type Items is array(Positive range <>) of Item;
    type Ring(The_Size : Positive) is
        record
            The_Top   : Natural := 0;
            The_Back  : Natural := 0;
            The_Mark  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Ring_Sequential_Bounded_Managed_Noniterator;
```

RING SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION


```
package body
Ring_Sequential_Bounded_Managed_Noniterator is

    procedure Copy (From_The_Ring : in     Ring;
                    To_The_Ring   : in out Ring) is
    begin
        if From_The_Ring.The_Back >
To_The_Ring.The_Size then
            raise Overflow;
        elsif From_The_Ring.The_Back = 0 then
            To_The_Ring.The_Top := 0;
            To_The_Ring.The_Back := 0;
            To_The_Ring.The_Mark := 0;
        else
            To_The_Ring.The_Items(1 ..
From_The_Ring.The_Back) :=
              From_The_Ring.The_Items(1 ..
From_The_Ring.The_Back);
            To_The_Ring.The_Top :=
From_The_Ring.The_Top;
            To_The_Ring.The_Back :=
From_The_Ring.The_Back;
            To_The_Ring.The_Mark :=
From_The_Ring.The_Mark;
        end if;
    end Copy;

    procedure Clear (The_Ring : in out Ring) is
    begin
        The_Ring.The_Top := 0;
        The_Ring.The_Back := 0;
        The_Ring.The_Mark := 0;
    end Clear;

    procedure Insert (The_Item    : in     Item;
                      In_The_Ring : in out Ring) is
    begin
        if In_The_Ring.The_Back = In_The_Ring.The_Size
then
            raise Overflow;
        elsif In_The_Ring.The_Back = 0 then
            In_The_Ring.The_Top := 1;
            In_The_Ring.The_Back := 1;
            In_The_Ring.The_Mark := 1;
            In_The_Ring.The_Items(1) := The_Item;
        else
            In_The_Ring.The_Items
              ((In_The_Ring.The_Top + 1) ..
(In_The_Ring.The_Back + 1)) :=
              In_The_Ring.The_Items(In_The_Ring.The_Top
..

In_The_Ring.The_Back);
            In_The_Ring.The_Items(In_The_Ring.The_Top)
:= The_Item;
            In_The_Ring.The_Back :=
In_The_Ring.The_Back + 1;
            if In_The_Ring.The_Mark >=
In_The_Ring.The_Top then
                In_The_Ring.The_Mark :=
In_The_Ring.The_Mark + 1;
            end if;
        end if;
    end Insert;

    procedure Pop(The_Ring : in out Ring) is
    begin
        if The_Ring.The_Back = 0 then
            raise Underflow;
        elsif The_Ring.The_Back = 1 then
            The_Ring.The_Top := 0;
            The_Ring.The_Back := 0;
            The_Ring.The_Mark := 0;
        else
            The_Ring.The_Items(The_Ring.The_Top ..
(The_Ring.The_Back - 1)) :=
              The_Ring.The_Items((The_Ring.The_Top + 1)
.. The_Ring.The_Back);
            The_Ring.The_Back := The_Ring.The_Back - 1;
            if The_Ring.The_Top > The_Ring.The_Back
then
                if The_Ring.The_Top = The_Ring.The_Mark
then
                    The_Ring.The_Mark := 1;
                end if;
                The_Ring.The_Top := 1;
            else
                if The_Ring.The_Mark > The_Ring.The_Top
then
                    The_Ring.The_Mark :=
The_Ring.The_Mark - 1;
                end if;
            end if;
        end if;
    end Pop;

    procedure Rotate (The_Ring         : in out Ring;
                      In_The_Direction : in
Direction) is
    begin
        if The_Ring.The_Back = 0 then
            raise Rotate_Error;
        elsif In_The_Direction = Forward then
            The_Ring.The_Top := The_Ring.The_Top + 1;
            if The_Ring.The_Top > The_Ring.The_Back
then
                The_Ring.The_Top := 1;
            end if;
        else
            The_Ring.The_Top := The_Ring.The_Top - 1;
            if The_Ring.The_Top = 0 then
                The_Ring.The_Top := The_Ring.The_Back;
            end if;
        end if;
    end Rotate;

    procedure Mark (The_Ring : in out Ring) is
    begin
        The_Ring.The_Mark := The_Ring.The_Top;
    end Mark;

    procedure Rotate_To_Mark (The_Ring : in out Ring)
is
    begin
        The_Ring.The_Top := The_Ring.The_Mark;
    end Rotate_To_Mark;

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural) is
    begin
        Result := Extent_Of(The_Ring);
    end Extent_Of;

    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Empty(The_Ring);
    end Is_Empty;

    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item) is
    begin
        Result := Top_Of(The_Ring);
    end Top_Of;

    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := At_Mark(The_Ring);
    end At_Mark;

--  end of modification

    function Is_Equal (Left  : in Ring;
                       Right : in Ring) return Boolean
is
        Left_Index  : Natural := Left.The_Top;
        Right_Index : Natural := Right.The_Top;
    begin
        if Left.The_Back /= Right.The_Back then
            return False;
        elsif Left.The_Items(Left_Index) /=
Right.The_Items(Right_Index) then
            return False;
        elsif (Left.The_Mark = Left_Index) and then
            (Right.The_Mark /= Right_Index) then
            return False;
        else
```

```
            Left_Index := Left_Index + 1;
            if Left_Index > Left.The_Back then
                Left_Index := 1;
            end if;
            Right_Index := Right_Index + 1;
            if Right_Index > Right.The_Back then
                Right_Index := 1;
            end if;
            while Left_Index /= Left.The_Top loop
                if Left.The_Items(Left_Index) /=
                   Right.The_Items(Right_Index) then
                    return False;
                elsif (Left.The_Mark = Left_Index) and
then
                    (Right.The_Mark /= Right_Index)
then
                    return False;
                else
                    Left_Index := Left_Index + 1;
                    if Left_Index > Left.The_Back then
                        Left_Index := 1;
                    end if;
                    Right_Index := Right_Index + 1;
                    if Right_Index > Right.The_Back
then
                        Right_Index := 1;
                    end if;
                end if;
            end loop;
            return (Right_Index = Right.The_Top);
        end if;
    exception
        when Constraint_Error =>
            return (Left.The_Top = Right.The_Top);
    end Is_Equal;

    function Extent_Of (The_Ring : in Ring) return
Natural is
    begin
        return The_Ring.The_Back;
    end Extent_Of;

    function Is_Empty (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Back = 0);
    end Is_Empty;

    function Top_Of (The_Ring : in Ring) return Item is
    begin
        return The_Ring.The_Items(The_Ring.The_Top);
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    function At_Mark (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = The_Ring.The_Mark);
    end At_Mark;

end Ring_Sequential_Bounded_Managed_Noniterator;
```

RING SEQUENTIAL BOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Ring_Sequential_Bounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Ring : Ring,
      To_The_Ring : Ring
    OUTPUT
      To_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Ring : Ring
    OUTPUT
      In_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate
  SPECIFICATION
    INPUT
      The_Ring : Ring,
      In_The_Direction : Direction
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate_To_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

END
IMPLEMENTATION ADA
Ring_Sequential_Bounded_Managed_Noniterator
END
```

RING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Ring_Sequential_Unbounded_Managed_Iterator is

    type Ring is limited private;

    type Direction is (Forward, Backward);

    procedure Copy             (From_The_Ring     : in
Ring;
                                To_The_Ring       : in out
Ring);
    procedure Clear            (The_Ring          : in out
Ring);
    procedure Insert           (The_Item          : in
Item;
                                In_The_Ring       : in out
Ring);
    procedure Pop              (The_Ring          : in out
Ring);
    procedure Rotate           (The_Ring          : in out
Ring;
                                In_The_Direction : in
Direction);
    procedure Mark             (The_Ring          : in out
Ring);
    procedure Rotate_To_Mark (The_Ring          : in out
Ring);

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
                         Result   : out Boolean);
    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural);
    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean);
    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item);
    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean);

--  end of modification

    function Is_Equal  (Left     : in Ring;
                        Right    : in Ring) return
Boolean;
    function Extent_Of (The_Ring : in Ring) return
Natural;
    function Is_Empty  (The_Ring : in Ring) return
Boolean;
    function Top_Of    (The_Ring : in Ring) return
Item;
    function At_Mark   (The_Ring : in Ring) return
Boolean;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out
Boolean);
    procedure Iterate (Over_The_Ring : in Ring);

    Overflow     : exception;
    Underflow    : exception;
    Rotate_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Ring is
        record
            The_Top  : Structure;
            The_Mark : Structure;
        end record;
end Ring_Sequential_Unbounded_Managed_Iterator;
```

RING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body Ring_Sequential_Unbounded_Managed_Iterator
is

    type Node is
        record
            Previous : Structure;
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        The_Node.Previous := null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return
Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new
Storage_Manager_Sequential
                                        (Item        =>
Node,
                                         Pointer     =>
Structure,
                          Free        => Free,
                          Set_Pointer => Set_Next,
                          Pointer_Of  => Next_Of);

    procedure Copy (From_The_Ring : in     Ring;
                    To_The_Ring   : in out Ring) is
        From_Index : Structure :=
From_The_Ring.The_Top;
        To_Index   : Structure;
    begin
        if To_The_Ring.The_Top /= null then
            To_The_Ring.The_Top.Previous.Next := null;
            Node_Manager.Free(To_The_Ring.The_Top);
        end if;
        if From_The_Ring.The_Top = null then
            To_The_Ring.The_Mark := null;
        else
            To_The_Ring.The_Top :=
Node_Manager.New_Item;
            To_The_Ring.The_Top.The_Item :=
From_Index.The_Item;
            To_Index := To_The_Ring.The_Top;
            if From_The_Ring.The_Mark = From_Index then
                To_The_Ring.The_Mark := To_Index;
            end if;
            From_Index := From_Index.Next;
            while From_Index /= From_The_Ring.The_Top
loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.Previous := To_Index;
                To_Index.Next.The_Item :=
From_Index.The_Item;
                To_Index := To_Index.Next;
                if From_The_Ring.The_Mark = From_Index
then
                    To_The_Ring.The_Mark := To_Index;
                end if;
                From_Index := From_Index.Next;
            end loop;
            To_The_Ring.The_Top.Previous := To_Index;
            To_Index.Next := To_The_Ring.The_Top;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Ring : in out Ring) is
    begin
        if The_Ring.The_Top /= null then
            The_Ring.The_Top.Previous.Next := null;
            Node_Manager.Free(The_Ring.The_Top);
            The_Ring.The_Mark := null;
        end if;
    end Clear;

    procedure Insert (The_Item    : in     Item;
                      In_The_Ring : in out Ring) is
        Temporary_Node : Structure;
    begin
        if In_The_Ring.The_Top = null then
            In_The_Ring.The_Top :=
Node_Manager.New_Item;
            In_The_Ring.The_Top.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.The_Item := The_Item;
            In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top;
            In_The_Ring.The_Mark :=
In_The_Ring.The_Top;
        else
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.Previous :=
In_The_Ring.The_Top.Previous;
            Temporary_Node.The_Item := The_Item;
            Temporary_Node.Next := In_The_Ring.The_Top;
            In_The_Ring.The_Top := Temporary_Node;
            In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The_Top;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Pop(The_Ring : in out Ring) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Ring.The_Top;
        if The_Ring.The_Top = The_Ring.The_Top.Next
then
            The_Ring.The_Top := null;
            The_Ring.The_Mark := null;
        else
            The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
            The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Previous;
            if The_Ring.The_Mark = The_Ring.The_Top
then
                The_Ring.The_Mark :=
The_Ring.The_Top.Next;
            end if;
            The_Ring.The_Top := The_Ring.The_Top.Next;
        end if;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
    exception
    when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Rotate (The_Ring         : in out Ring;
                      In_The_Direction : in
Direction) is
    begin
        if In_The_Direction = Forward then
            The_Ring.The_Top := The_Ring.The_Top.Next;
        else
            The_Ring.The_Top :=
The_Ring.The_Top.Previous;
        end if;
    exception
        when Constraint_Error =>
            raise Rotate_Error;
    end Rotate;

    procedure Mark (The_Ring : in out Ring) is
    begin
        The_Ring.The_Mark := The_Ring.The_Top;
    end Mark;

    procedure Rotate_To_Mark (The_Ring : in out Ring)
is
    begin
        The_Ring.The_Top := The_Ring.The_Mark;
    end Rotate_To_Mark;

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
```

```
                          Result    : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Ring : in Ring;
                          Result    : out Natural) is
    begin
        Result := Extent_Of(The_Ring);
    end Extent_Of;

    procedure Is_Empty  (The_Ring : in Ring;
                          Result    : out Boolean) is
    begin
        Result := Is_Empty(The_Ring);
    end Is_Empty;

    procedure Top_Of    (The_Ring : in Ring;
                          Result    : out Item) is
    begin
        Result := Top_Of(The_Ring);
    end Top_Of;

    procedure At_Mark   (The_Ring : in Ring;
                          Result    : out Boolean) is
    begin
        Result := At_Mark(The_Ring);
    end At_Mark;

--  end of modification

    function Is_Equal (Left  : in Ring;
                       Right : in Ring) return Boolean
is
        Left_Index  : Structure := Left.The_Top;
        Right_Index : Structure := Right.The_Top;
    begin
        if Left_Index.The_Item /= Right_Index.The_Item
then
            return False;
        elsif (Left.The_Mark = Left_Index) and then
            (Right.The_Mark /= Right_Index) then
            return False;
        else
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
            while Left_Index /= Left.The_Top loop
                if Left_Index.The_Item /=
Right_Index.The_Item then
                    return False;
                elsif (Left.The_Mark = Left_Index) and
then
                    (Right.The_Mark /= Right_Index) then
                    return False;
                else
                    Left_Index := Left_Index.Next;
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            return (Right_Index = Right.The_Top);
        end if;
    exception
        when Constraint_Error =>
            return (Left.The_Top = Right.The_Top);
    end Is_Equal;

    function Extent_Of (The_Ring : in Ring) return
Natural is
        Count : Natural   := 0;
        Index : Structure := The_Ring.The_Top;
    begin
        Index := Index.Next;
        Count := Count + 1;
        while Index /= The_Ring.The_Top loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    exception
        when Constraint_Error =>
            return 0;
    end Extent_Of;

    function Is_Empty (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = null);
    end Is_Empty;

    function Top_Of (The_Ring : in Ring) return Item is
    begin
        return The_Ring.The_Top.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    function At_Mark (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = The_Ring.The_Mark);
    end At_Mark;

    procedure Iterate (Over_The_Ring : in Ring) is
        The_Iterator : Structure :=
Over_The_Ring.The_Top;
        Continue     : Boolean;
    begin
        if The_Iterator /= null then
            Process(The_Iterator.The_Item, Continue);
            if Continue then
                The_Iterator := The_Iterator.Next;
                while not (The_Iterator =
Over_The_Ring.The_Top) loop
                    Process(The_Iterator.The_Item,
Continue);
                    exit when not Continue;
                    The_Iterator := The_Iterator.Next;
                end loop;
            end if;
        end if;
    end Iterate;

end Ring_Sequential_Unbounded_Managed_Iterator;
```

RING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Ring_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Ring : Ring,
      To_The_Ring : Ring
    OUTPUT
      To_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Ring : Ring
    OUTPUT
      In_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate
  SPECIFICATION
    INPUT
      The_Ring : Ring,
      In_The_Direction : Direction
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate_To_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Ring,
      Right : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR At_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
    INPUT
      Over_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

END
IMPLEMENTATION ADA
Ring_Sequential_Unbounded_Managed_Iterator
END
```

RING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Ring_Sequential_Unbounded_Managed_Noniterator
is

    type Ring is limited private;

    type Direction is (Forward, Backward);

    procedure Copy            (From_The_Ring    : in
Ring;
                               To_The_Ring      : in out
Ring);
    procedure Clear           (The_Ring         : in out
Ring);
    procedure Insert          (The_Item         : in
Item;
                               In_The_Ring      : in out
Ring);
    procedure Pop             (The_Ring         : in out
Ring);
    procedure Rotate          (The_Ring         : in out
Ring;
                               In_The_Direction : in
Direction);
    procedure Mark            (The_Ring         : in out
Ring);
    procedure Rotate_To_Mark (The_Ring          : in out
Ring);

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left      : in Ring;
                         Right     : in Ring;
                         Result    : out Boolean);
    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural);
    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean);
    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item);
    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean);

--  end of modification

    function Is_Equal  (Left      : in Ring;
                        Right     : in Ring) return
Boolean;
    function Extent_Of (The_Ring : in Ring) return
Natural;
    function Is_Empty  (The_Ring : in Ring) return
Boolean;
    function Top_Of    (The_Ring : in Ring) return
Item;
    function At_Mark   (The_Ring : in Ring) return
Boolean;

    Overflow     : exception;
    Underflow    : exception;
    Rotate_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Ring is
        record
            The_Top  : Structure;
            The_Mark : Structure;
        end record;
end Ring_Sequential_Unbounded_Managed_Noniterator;
```

RING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Ring_Sequential_Unbounded_Managed_Noniterator is

    type Node is
        record
            Previous : Structure;
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        The_Node.Previous := null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Structure) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return
Structure is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new
Storage_Manager_Sequential
                                        (Item        =>
Node,
                                         Pointer     =>
Structure,
                           Free        => Free,
                           Set_Pointer => Set_Next,
                           Pointer_Of  => Next_Of);

    procedure Copy (From_The_Ring : in     Ring;
                    To_The_Ring   : in out Ring) is
        From_Index : Structure :=
From_The_Ring.The_Top;
        To_Index   : Structure;
    begin
        if To_The_Ring.The_Top /= null then
            To_The_Ring.The_Top.Previous.Next := null;
            Node_Manager.Free(To_The_Ring.The_Top);
        end if;
        if From_The_Ring.The_Top = null then
            To_The_Ring.The_Mark := null;
        else
            To_The_Ring.The_Top :=
Node_Manager.New_Item;
            To_The_Ring.The_Top.The_Item :=
From_Index.The_Item;
            To_Index := To_The_Ring.The_Top;
            if From_The_Ring.The_Mark = From_Index then
                To_The_Ring.The_Mark := To_Index;
            end if;
            From_Index := From_Index.Next;
            while From_Index /= From_The_Ring.The_Top
loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index.Next.Previous := To_Index;
                To_Index.Next.The_Item :=
From_Index.The_Item;
                To_Index := To_Index.Next;
                if From_The_Ring.The_Mark = From_Index
then
                    To_The_Ring.The_Mark := To_Index;
                end if;
                From_Index := From_Index.Next;
            end loop;
            To_The_Ring.The_Top.Previous := To_Index;
            To_Index.Next := To_The_Ring.The_Top;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Ring : in out Ring) is
    begin
        if The_Ring.The_Top /= null then
            The_Ring.The_Top.Previous.Next := null;
            Node_Manager.Free(The_Ring.The_Top);
            The_Ring.The_Mark := null;
        end if;
    end Clear;

    procedure Insert (The_Item    : in     Item;
                      In_The_Ring : in out Ring) is
        Temporary_Node : Structure;
    begin
        if In_The_Ring.The_Top = null then
            In_The_Ring.The_Top :=
Node_Manager.New_Item;
            In_The_Ring.The_Top.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.The_Item := The_Item;
            In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top;
            In_The_Ring.The_Mark :=
In_The_Ring.The_Top;
        else
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.Previous :=
In_The_Ring.The_Top.Previous;
            Temporary_Node.The_Item := The_Item;
            Temporary_Node.Next := In_The_Ring.The_Top;
            In_The_Ring.The_Top := Temporary_Node;
            In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The_Top;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Pop(The_Ring : in out Ring) is
        Temporary_Node : Structure;
    begin
        Temporary_Node := The_Ring.The_Top;
        if The_Ring.The_Top = The_Ring.The_Top.Next
then
            The_Ring.The_Top := null;
            The_Ring.The_Mark := null;
        else
            The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
            The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Previous;
            if The_Ring.The_Mark = The_Ring.The_Top
then
                The_Ring.The_Mark :=
The_Ring.The_Top.Next;
            end if;
            The_Ring.The_Top := The_Ring.The_Top.Next;
        end if;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
    exception
    when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Rotate (The_Ring         : in out Ring;
                      In_The_Direction : in
Direction) is
    begin
        if In_The_Direction = Forward then
            The_Ring.The_Top := The_Ring.The_Top.Next;
        else
            The_Ring.The_Top :=
The_Ring.The_Top.Previous;
        end if;
    exception
        when Constraint_Error =>
            raise Rotate_Error;
    end Rotate;

    procedure Mark (The_Ring : in out Ring) is
    begin
        The_Ring.The_Mark := The_Ring.The_Top;
    end Mark;

    procedure Rotate_To_Mark (The_Ring : in out Ring)
is
    begin
        The_Ring.The_Top := The_Ring.The_Mark;
    end Rotate_To_Mark;

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal (Left     : in Ring;
                        Right    : in Ring;
```

```
                         Result   : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural) is
    begin
        Result := Extent_Of(The_Ring);
    end Extent_Of;

    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Empty(The_Ring);
    end Is_Empty;

    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item) is
    begin
        Result := Top_Of(The_Ring);
    end Top_Of;

    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := At_Mark(The_Ring);
    end At_Mark;

--  end of modification

    function Is_Equal (Left  : in Ring;
                       Right : in Ring) return Boolean
is
        Left_Index  : Structure := Left.The_Top;
        Right_Index : Structure := Right.The_Top;
    begin
        if Left_Index.The_Item /= Right_Index.The_Item
then
            return False;
        elsif (Left.The_Mark = Left_Index) and then
          (Right.The_Mark /= Right_Index) then
            return False;
        else
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
            while Left_Index /= Left.The_Top loop
                if Left_Index.The_Item /=
Right_Index.The_Item then
                    return False;
                elsif (Left.The_Mark = Left_Index) and
then
                  (Right.The_Mark /= Right_Index) then
                    return False;
                else
                    Left_Index := Left_Index.Next;
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            return (Right_Index = Right.The_Top);
        end if;
    exception
        when Constraint_Error =>
            return (Left.The_Top = Right.The_Top);
    end Is_Equal;

    function Extent_Of (The_Ring : in Ring) return
Natural is
        Count : Natural   := 0;
        Index : Structure := The_Ring.The_Top;
    begin
        Index := Index.Next;
        Count := Count + 1;
        while Index /= The_Ring.The_Top loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    exception
        when Constraint_Error =>
            return 0;
    end Extent_Of;

    function Is_Empty (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = null);
    end Is_Empty;

    function Top_Of (The_Ring : in Ring) return Item is
    begin
        return The_Ring.The_Top.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    function At_Mark (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = The_Ring.The_Mark);
    end At_Mark;

end Ring_Sequential_Unbounded_Managed_Noniterator;
```

RING SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Ring_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Ring : Ring,
      To_The_Ring : Ring
    OUTPUT
      To_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Ring : Ring
    OUTPUT
      In_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate
  SPECIFICATION
    INPUT
      The_Ring : Ring,
      In_The_Direction : Direction
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate_To_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Ring,
      Right : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR At_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

END
IMPLEMENTATION ADA
Ring_Sequential_Unbounded_Managed_Noniterator
END
```

RING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Ring_Sequential_Unbounded_Unmanaged_Iterator is

    type Ring is limited private;

    type Direction is (Forward, Backward);

    procedure Copy            (From_The_Ring    : in
Ring;
                               To_The_Ring      : in out
Ring);
    procedure Clear           (The_Ring         : in out
Ring);
    procedure Insert          (The_Item         : in
Item;
                               In_The_Ring      : in out
Ring);
    procedure Pop             (The_Ring         : in out
Ring);
    procedure Rotate          (The_Ring         : in out
Ring;
                               In_The_Direction : in
Direction);
    procedure Mark            (The_Ring         : in out
Ring);
    procedure Rotate_To_Mark  (The_Ring         : in out
Ring);

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
                         Result   : out Boolean);
    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural);
    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean);
    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item);
    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean);

--  end of modification

    function Is_Equal  (Left     : in Ring;
                        Right    : in Ring) return
Boolean;
    function Extent_Of (The_Ring : in Ring) return
Natural;
    function Is_Empty  (The_Ring : in Ring) return
Boolean;
    function Top_Of    (The_Ring : in Ring) return
Item;
    function At_Mark   (The_Ring : in Ring) return
Boolean;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out
Boolean);
    procedure Iterate (Over_The_Ring : in Ring);

    Overflow     : exception;
    Underflow    : exception;
    Rotate_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Ring is
        record
            The_Top  : Structure;
            The_Mark : Structure;
        end record;
end Ring_Sequential_Unbounded_Unmanaged_Iterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Ring_Sequential_Unbounded_Unmanaged_Iterator is

    type Node is
        record
            Previous : Structure;
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Ring : in     Ring;
                    To_The_Ring   : in out Ring) is
        From_Index : Structure :=
From_The_Ring.The_Top;
        To_Index   : Structure;
    begin
        if From_The_Ring.The_Top = null then
            To_The_Ring.The_Top := null;
            To_The_Ring.The_Mark := null;
        else
            To_The_Ring.The_Top := new Node'(Previous
=> null,
                                             The_Item
=> From_Index.The_Item,
                                             Next
=> null);
            To_Index := To_The_Ring.The_Top;
            if From_The_Ring.The_Mark = From_Index then
                To_The_Ring.The_Mark := To_Index;
            end if;
            From_Index := From_Index.Next;
            while From_Index /= From_The_Ring.The_Top
loop
                To_Index.Next := new Node'(Previous =>
To_Index,
                                           The_Item =>
From_Index.The_Item,
                                           Next     =>
null);
                To_Index := To_Index.Next;
                if From_The_Ring.The_Mark = From_Index
then
                    To_The_Ring.The_Mark := To_Index;
                end if;
                From_Index := From_Index.Next;
            end loop;
            To_The_Ring.The_Top.Previous := To_Index;
            To_Index.Next := To_The_Ring.The_Top;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Ring : in out Ring) is
    begin
        The_Ring := Ring'(The_Top  => null,
                          The_Mark => null);
    end Clear;

    procedure Insert (The_Item    : in     Item;
                      In_The_Ring : in out Ring) is
    begin
        if In_The_Ring.The_Top = null then
            In_The_Ring.The_Top := new Node'(Previous
=> null,
                                             The_Item
=> The_Item,
                                             Next
=> null);
            In_The_Ring.The_Top.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top;
            In_The_Ring.The_Mark :=
In_The_Ring.The_Top;
        else
            In_The_Ring.The_Top :=
              new Node'(Previous =>
In_The_Ring.The_Top.Previous,
                        The_Item => The_Item,
                        Next     =>
In_The_Ring.The_Top);
            In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The_Top;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Pop(The_Ring : in out Ring) is
    begin
        if The_Ring.The_Top = The_Ring.The_Top.Next
then
            The_Ring.The_Top := null;
            The_Ring.The_Mark := null;
        else
            The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
            The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Previous;
            if The_Ring.The_Mark = The_Ring.The_Top
then
                The_Ring.The_Mark :=
The_Ring.The_Top.Next;
            end if;
            The_Ring.The_Top := The_Ring.The_Top.Next;
        end if;
    exception
    when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Rotate (The_Ring         : in out Ring;
                      In_The_Direction : in
Direction) is
    begin
        if In_The_Direction = Forward then
            The_Ring.The_Top := The_Ring.The_Top.Next;
        else
            The_Ring.The_Top :=
The_Ring.The_Top.Previous;
        end if;
    exception
        when Constraint_Error =>
            raise Rotate_Error;
    end Rotate;

    procedure Mark (The_Ring : in out Ring) is
    begin
        The_Ring.The_Mark := The_Ring.The_Top;
    end Mark;

    procedure Rotate_To_Mark (The_Ring : in out Ring)
is
    begin
        The_Ring.The_Top := The_Ring.The_Mark;
    end Rotate_To_Mark;

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural) is
    begin
        Result := Extent_Of(The_Ring);
    end Extent_Of;

    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Empty(The_Ring);
    end Is_Empty;

    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item) is
    begin
        Result := Top_Of(The_Ring);
    end Top_Of;

    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := At_Mark(The_Ring);
    end At_Mark;

--  end of modification
```

```
    function Is_Equal (Left  : in Ring;
                       Right : in Ring) return Boolean
is
        Left_Index  : Structure := Left.The_Top;
        Right_Index : Structure := Right.The_Top;
    begin
        if Left_Index.The_Item /= Right_Index.The_Item
then
            return False;
        elsif (Left.The_Mark = Left_Index) and then
           (Right.The_Mark /= Right_Index) then
            return False;
        else
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
            while Left_Index /= Left.The_Top loop
                if Left_Index.The_Item /=
Right_Index.The_Item then
                    return False;
                elsif (Left.The_Mark = Left_Index) and
then
                   (Right.The_Mark /= Right_Index) then
                    return False;
                else
                    Left_Index := Left_Index.Next;
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            return (Right_Index = Right.The_Top);
        end if;
    exception
        when Constraint_Error =>
            return (Left.The_Top = Right.The_Top);
    end Is_Equal;

    function Extent_Of (The_Ring : in Ring) return
Natural is
        Count : Natural   := 0;
        Index : Structure := The_Ring.The_Top;
    begin
        Index := Index.Next;
        Count := Count + 1;
        while Index /= The_Ring.The_Top loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    exception
        when Constraint_Error =>
            return 0;
    end Extent_Of;

    function Is_Empty (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = null);
    end Is_Empty;

    function Top_Of (The_Ring : in Ring) return Item is
    begin
        return The_Ring.The_Top.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    function At_Mark (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = The_Ring.The_Mark);
    end At_Mark;

    procedure Iterate (Over_The_Ring : in Ring) is
        The_Iterator : Structure :=
Over_The_Ring.The_Top;
        Continue     : Boolean;
    begin
        if The_Iterator /= null then
            Process(The_Iterator.The_Item, Continue);
            if Continue then
                The_Iterator := The_Iterator.Next;
                while not (The_Iterator =
Over_The_Ring.The_Top) loop
                    Process(The_Iterator.The_Item,
Continue);
                    exit when not Continue;
                    The_Iterator := The_Iterator.Next;
                end loop;
            end if;
        end if;
    end Iterate;

end Ring_Sequential_Unbounded_Unmanaged_Iterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Ring_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Ring : Ring,
      To_The_Ring : Ring
    OUTPUT
      To_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Ring : Ring
    OUTPUT
      In_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate
  SPECIFICATION
    INPUT
      The_Ring : Ring,
      In_The_Direction : Direction
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate_To_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Ring,
      Right : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR At_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
    INPUT
      Over_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

END
IMPLEMENTATION ADA
Ring_Sequential_Unbounded_Unmanaged_Iterator
END
```

RING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Ring_Sequential_Unbounded_Unmanaged_Noniterator
is

    type Ring is limited private;

    type Direction is (Forward, Backward);

    procedure Copy             (From_The_Ring    : in
Ring;
                                To_The_Ring      : in out
Ring);
    procedure Clear            (The_Ring         : in out
Ring);
    procedure Insert           (The_Item         : in
Item;
                                In_The_Ring      : in out
Ring);
    procedure Pop              (The_Ring         : in out
Ring);
    procedure Rotate           (The_Ring         : in out
Ring;
                                In_The_Direction : in
Direction);
    procedure Mark             (The_Ring         : in out
Ring);
    procedure Rotate_To_Mark (The_Ring           : in out
Ring);

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
                         Result   : out Boolean);
    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural);
    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean);
    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item);
    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean);

--  end of modification

    function Is_Equal  (Left     : in Ring;
                        Right    : in Ring) return
Boolean;
    function Extent_Of (The_Ring : in Ring) return
Natural;
    function Is_Empty  (The_Ring : in Ring) return
Boolean;
    function Top_Of    (The_Ring : in Ring) return
Item;
    function At_Mark   (The_Ring : in Ring) return
Boolean;

    Overflow     : exception;
    Underflow    : exception;
    Rotate_Error : exception;

private
    type Node;
    type Structure is access Node;
    type Ring is
        record
            The_Top  : Structure;
            The_Mark : Structure;
        end record;
end Ring_Sequential_Unbounded_Unmanaged_Noniterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Ring_Sequential_Unbounded_Unmanaged_Noniterator is

    type Node is
        record
            Previous : Structure;
            The_Item : Item;
            Next     : Structure;
        end record;

    procedure Copy (From_The_Ring : in     Ring;
                    To_The_Ring   : in out Ring) is
        From_Index : Structure :=
From_The_Ring.The_Top;
        To_Index   : Structure;
    begin
        if From_The_Ring.The_Top = null then
            To_The_Ring.The_Top := null;
            To_The_Ring.The_Mark := null;
        else
            To_The_Ring.The_Top := new Node'(Previous
=> null,
                                                The_Item
=> From_Index.The_Item,
                                                Next
=> null);
            To_Index := To_The_Ring.The_Top;
            if From_The_Ring.The_Mark = From_Index then
                To_The_Ring.The_Mark := To_Index;
            end if;
            From_Index := From_Index.Next;
            while From_Index /= From_The_Ring.The_Top
loop
                To_Index.Next := new Node'(Previous =>
To_Index,
                                            The_Item =>
From_Index.The_Item,
                                            Next     =>
null);
                To_Index := To_Index.Next;
                if From_The_Ring.The_Mark = From_Index
then
                    To_The_Ring.The_Mark := To_Index;
                end if;
                From_Index := From_Index.Next;
            end loop;
            To_The_Ring.The_Top.Previous := To_Index;
            To_Index.Next := To_The_Ring.The_Top;
        end if;
    exception
       when Storage_Error =>
          raise Overflow;
    end Copy;

    procedure Clear (The_Ring : in out Ring) is
    begin
        The_Ring := Ring'(The_Top  => null,
                          The_Mark => null);
    end Clear;

    procedure Insert (The_Item    : in     Item;
                      In_The_Ring : in out Ring) is
    begin
        if In_The_Ring.The_Top = null then
            In_The_Ring.The_Top := new Node'(Previous
=> null,
                                                The_Item
=> The_Item,
                                                Next
=> null);
            In_The_Ring.The_Top.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.Next :=
In_The_Ring.The_Top;
            In_The_Ring.The_Mark :=
In_The_Ring.The_Top;
        else
            In_The_Ring.The_Top :=
              new Node'(Previous =>
In_The_Ring.The_Top.Previous,
                        The_Item => The_Item,
                        Next     =>
In_The_Ring.The_Top);
            In_The_Ring.The_Top.Next.Previous :=
In_The_Ring.The_Top;
            In_The_Ring.The_Top.Previous.Next :=
In_The_Ring.The_Top;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Pop(The_Ring : in out Ring) is
    begin
        if The_Ring.The_Top = The_Ring.The_Top.Next
then
            The_Ring.The_Top := null;
            The_Ring.The_Mark := null;
        else
            The_Ring.The_Top.Previous.Next :=
The_Ring.The_Top.Next;
            The_Ring.The_Top.Next.Previous :=
The_Ring.The_Top.Previous;
            if The_Ring.The_Mark = The_Ring.The_Top
then
                The_Ring.The_Mark :=
The_Ring.The_Top.Next;
            end if;
            The_Ring.The_Top := The_Ring.The_Top.Next;
        end if;
    exception
    when Constraint_Error =>
            raise Underflow;
    end Pop;

    procedure Rotate (The_Ring         : in out Ring;
                      In_The_Direction : in
Direction) is
    begin
        if In_The_Direction = Forward then
            The_Ring.The_Top := The_Ring.The_Top.Next;
        else
            The_Ring.The_Top :=
The_Ring.The_Top.Previous;
        end if;
    exception
        when Constraint_Error =>
            raise Rotate_Error;
    end Rotate;

    procedure Mark (The_Ring : in out Ring) is
    begin
        The_Ring.The_Mark := The_Ring.The_Top;
    end Mark;

    procedure Rotate_To_Mark (The_Ring : in out Ring)
is
    begin
        The_Ring.The_Top := The_Ring.The_Mark;
    end Rotate_To_Mark;

--  modified by Tuan Nguyen
--  10 January 1996
--  adding procedures to replace functions

    procedure Is_Equal  (Left     : in Ring;
                         Right    : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of (The_Ring : in Ring;
                         Result   : out Natural) is
    begin
        Result := Extent_Of(The_Ring);
    end Extent_Of;

    procedure Is_Empty  (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := Is_Empty(The_Ring);
    end Is_Empty;

    procedure Top_Of    (The_Ring : in Ring;
                         Result   : out Item) is
    begin
        Result := Top_Of(The_Ring);
    end Top_Of;

    procedure At_Mark   (The_Ring : in Ring;
                         Result   : out Boolean) is
    begin
        Result := At_Mark(The_Ring);
    end At_Mark;

--  end of modification
```

```
    function Is_Equal (Left  : in Ring;
                       Right : in Ring) return Boolean
is
        Left_Index  : Structure := Left.The_Top;
        Right_Index : Structure := Right.The_Top;
    begin
        if Left_Index.The_Item /= Right_Index.The_Item
then
            return False;
        elsif (Left.The_Mark = Left_Index) and then
           (Right.The_Mark /= Right_Index) then
            return False;
        else
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
            while Left_Index /= Left.The_Top loop
                if Left_Index.The_Item /=
Right_Index.The_Item then
                    return False;
                elsif (Left.The_Mark = Left_Index) and
then
                    (Right.The_Mark /= Right_Index) then
                    return False;
                else
                    Left_Index := Left_Index.Next;
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            return (Right_Index = Right.The_Top);
        end if;
    exception
        when Constraint_Error =>
            return (Left.The_Top = Right.The_Top);
    end Is_Equal;

    function Extent_Of (The_Ring : in Ring) return
Natural is
        Count : Natural   := 0;
        Index : Structure := The_Ring.The_Top;
    begin
        Index := Index.Next;
        Count := Count + 1;
        while Index /= The_Ring.The_Top loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    exception
        when Constraint_Error =>
            return 0;
    end Extent_Of;

    function Is_Empty (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = null);
    end Is_Empty;

    function Top_Of (The_Ring : in Ring) return Item is
    begin
        return The_Ring.The_Top.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    function At_Mark (The_Ring : in Ring) return
Boolean is
    begin
        return (The_Ring.The_Top = The_Ring.The_Mark);
    end At_Mark;

end Ring_Sequential_Unbounded_Unmanaged_Noniterator;
```

RING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE Ring_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Ring : Ring,
      To_The_Ring : Ring
    OUTPUT
      To_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Item : Item,
      In_The_Ring : Ring
    OUTPUT
      In_The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate
  SPECIFICATION
    INPUT
      The_Ring : Ring,
      In_The_Direction : Direction
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Rotate_To_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      The_Ring : Ring
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Ring,
      Right : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

  OPERATOR At_Mark
  SPECIFICATION
    INPUT
      The_Ring : Ring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow, Rotate_Error
  END

END
IMPLEMENTATION ADA
Ring_Sequential_Unbounded_Unmanaged_Noniterator
END
```

SETS OBJ3 SPECIFICATIONS

```
obj SET[X :: TRIV] is sort Set .
   protecting NAT .

*** constructors

   op create          :             -> Set .
   op copy            :     Set Set -> Set .
   op clear           :     Set -> Set .
   op add             :     Elt Set -> Set .
   op remove          :     Elt Set -> Set .
   op union           : Set Set Set -> Set .
   op intersection    : Set Set Set -> Set .
   op difference      : Set Set Set -> Set .

*** accessors

   op isequal         : Set Set -> Bool .
   op extentof        :     Set -> Nat .
   op isempty         :     Set -> Bool .
   op isamember       : Elt Set -> Bool .
   op isasubset       : Set Set -> Bool .
   op isapropersubset : Set Set -> Bool .

*** exceptions

   op overflow        : -> Set .
   op itemisinset     : -> Set .
   op itemisnotinset  : -> Set .

*** variables declaration

   var S S1 S2 : Set .
   var E E1 : Elt .

*** axioms

   eq copy(S,S1) = S .

   eq clear(S) = create .

   eq remove(E,create) = itemisnotinset .
   eq remove(E,add(E1,S1)) = if E == E1 then S1 else
add(E1,remove(E,S1)) fi .

   eq union(S,create,S1) = S .
   eq union(S,add(E1,S1),S2) = if isamember(E1,S) then union(S,S1,S2)
else add(E1,union(S,S1,S2)) fi .

   eq intersection(S,create,S1) = create .
   eq intersection(S,add(E1,S1),S2) = if isamember(E1,S) then
add(E1,intersection(S,S1,S2)) else intersection(S,S1,S2) fi .

   eq difference(create,S,S1) = S .
   eq difference(S,create,S1) = S .
   eq difference(S,add(E1,S1),S2) = if isamember(E1,S) then
difference(remove(E1,S),S1,S2) else add(E1,difference(S,S1,S2)) fi .

   eq isequal(S,S1) = S == S1 .

   eq extentof(create) = 0 .
   eq extentof(add(E,S)) = 1 + extentof(S) .

   eq isamember(E,create) = false .
   eq isamember(E,add(E1,S1)) = E == E1 or isamember(E,S1) .

   eq isasubset(create,S) = true .
   eq isasubset(add(E,S),S1) = if isamember(E,S1) then
isasubset(S,remove(E,S1)) else false fi .

   eq isapropersubset(S,S1) = isasubset(S,S1) and extentof(S1) >
extentof(S) .

endo
```

SET PROFILE CODES

OPERATORS	*SIGNATURES*	*PROFILE CODES*
COPY	A B -> B	3211
CLEAR	A -> A	2201
ADD	A B -> B	3211
REMOVE	A B -> B	3211
UNION	A B C -> C	4231
INTERSECTION	A B C -> C	4231
DIFFERENCE	A B C -> C	4231
IS_EQUAL	A B -> C	330
EXTENT_OF	A -> B	220
IS_EMPTY	A -> B	220
IS_A_MEMBER	A B -> C	330
IS_A_SUBSET	A B -> C	330
IS_A_PROPER_SUBSET	A B -> C	330

SET OF PROFILE: {4231,3211,2201,330,220}

SET SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Set_Simple_Sequential_Bounded_Managed_Iterator is

    type Set(The_Size : Positive) is limited private;

    procedure Copy          (From_The_Set : in      Set;
                             To_The_Set   : in out Set);
    procedure Clear         (The_Set      : in out Set);
    procedure Add           (The_Item     : in      Item;
                             To_The_Set   : in out Set);
    procedure Remove        (The_Item     : in      Item;
                             From_The_Set : in out Set);
    procedure Union         (Of_The_Set   : in      Set;
                             And_The_Set  : in      Set;
                             To_The_Set   : in out Set);
    procedure Intersection  (Of_The_Set   : in      Set;
                             And_The_Set  : in      Set;
                             To_The_Set   : in out Set);
    procedure Difference    (Of_The_Set   : in      Set;
                             And_The_Set  : in      Set;
                             To_The_Set   : in out Set);

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal            (Left        : in Set;
                                   Right       : in Set;
                                   Result      : out Boolean);
    procedure Extent_Of           (The_Set     : in Set;
                                   Result      : out Natural);
    procedure Is_Empty            (The_Set     : in Set;
                                   Result      : out Boolean);
    procedure Is_A_Member         (The_Item    : in Item;
                                   Of_The_Set  : in Set;
                                   Result      : out Boolean);
    procedure Is_A_Subset         (Left        : in Set;
                                   Right       : in Set;
                                   Result      : out Boolean);
    procedure Is_A_Proper_Subset  (Left        : in Set;
                                   Right       : in Set;
                                   Result      : out Boolean);

--  end of modification

    function Is_Equal            (Left        : in Set;
                                  Right       : in Set)  return Boolean;
    function Extent_Of           (The_Set     : in Set)  return Natural;
    function Is_Empty            (The_Set     : in Set)  return Boolean;
    function Is_A_Member         (The_Item    : in Item;
                                  Of_The_Set  : in Set)  return Boolean;
    function Is_A_Subset         (Left        : in Set;
                                  Right       : in Set)  return Boolean;
    function Is_A_Proper_Subset  (Left        : in Set;
                                  Right       : in Set)  return Boolean;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Set : in Set);

    Overflow           : exception;
    Item_Is_In_Set     : exception;
    Item_Is_Not_In_Set : exception;

private
    type Items is array(Positive range <>) of Item;
    type Set(The_Size : Positive) is
        record
            The_Back  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Set_Simple_Sequential_Bounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Set_Simple_Sequential_Bounded_Managed_Iterator is

    procedure Copy (From_The_Set : in     Set;
                    To_The_Set   : in out Set) is
    begin
        if From_The_Set.The_Back > To_The_Set.The_Size then
            raise Overflow;
        else
            To_The_Set.The_Items(1 .. From_The_Set.The_Back) :=
              From_The_Set.The_Items(1 .. From_The_Set.The_Back);
            To_The_Set.The_Back := From_The_Set.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Set : in out Set) is
    begin
        The_Set.The_Back := 0;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Set : in out Set) is
    begin
        for Index in 1 .. To_The_Set.The_Back loop
            if The_Item = To_The_Set.The_Items(Index) then
                raise Item_Is_In_Set;
            end if;
        end loop;
        To_The_Set.The_Items(To_The_Set.The_Back + 1) := The_Item;
        To_The_Set.The_Back := To_The_Set.The_Back + 1;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Set : in out Set) is
    begin
        for Index in 1 .. From_The_Set.The_Back loop
            if The_Item = From_The_Set.The_Items(Index) then
                From_The_Set.The_Items(Index .. (From_The_Set.The_Back
- 1)) :=
                  From_The_Set.The_Items((Index + 1) ..
From_The_Set.The_Back);
                From_The_Set.The_Back := From_The_Set.The_Back - 1;
                return;
            end if;
        end loop;
        raise Item_Is_Not_In_Set;
    end Remove;

    procedure Union (Of_The_Set : in     Set;
                     And_The_Set: in     Set;
                     To_The_Set : in out Set) is
        To_Index : Natural;
        To_Back  : Natural;
    begin
        To_The_Set.The_Items(1 .. Of_The_Set.The_Back) :=
          Of_The_Set.The_Items(1 .. Of_The_Set.The_Back);
        To_The_Set.The_Back := Of_The_Set.The_Back;
        To_Back := To_The_Set.The_Back;
        for And_Index in 1 .. And_The_Set.The_Back loop
            To_Index := To_Back;
            while To_Index > 0 loop
                if To_The_Set.The_Items(To_Index) =
                   And_The_Set.The_Items(And_Index) then
                    exit;
                else
                    To_Index := To_Index - 1;
                end if;
            end loop;
            if To_Index = 0 then
                To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
                  And_The_Set.The_Items(And_Index);
                To_The_Set.The_Back := To_The_Set.The_Back + 1;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Set  : in     Set;
                            And_The_Set : in     Set;
                            To_The_Set  : in out Set) is
        And_Index : Natural;
    begin
        To_The_Set.The_Back := 0;
        for Of_Index in 1 .. Of_The_Set.The_Back loop
            And_Index := And_The_Set.The_Back;
            while And_Index > 0 loop
                if Of_The_Set.The_Items(Of_Index) =
                   And_The_Set.The_Items(And_Index) then
                    To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
                      Of_The_Set.The_Items(Of_Index);
                    To_The_Set.The_Back := To_The_Set.The_Back + 1;
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Set  : in     Set;
                          And_The_Set : in     Set;
                          To_The_Set  : in out Set) is
        And_Index : Natural;
    begin
        To_The_Set.The_Back := 0;
        for Of_Index in 1 .. Of_The_Set.The_Back loop
            And_Index := And_The_Set.The_Back;
            while And_Index > 0 loop
                if Of_The_Set.The_Items(Of_Index) =
                   And_The_Set.The_Items(And_Index) then
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
            if And_Index = 0 then
                To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
                  Of_The_Set.The_Items(Of_Index);
                To_The_Set.The_Back := To_The_Set.The_Back + 1;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Difference;

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal            (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of           (The_Set    : in Set;
                                   Result     : out Natural) is
    begin
        Result := Extent_Of(The_Set);
    end Extent_Of;

    procedure Is_Empty            (The_Set    : in Set;
                                   Result     : out Boolean) is
    begin
        Result := Is_Empty(The_Set);
    end Is_Empty;

    procedure Is_A_Member         (The_Item   : in Item;
                                   Of_The_Set : in Set;
                                   Result     : out Boolean) is
    begin
        Result := Is_A_Member(The_Item,Of_The_Set);
    end Is_A_Member;

    procedure Is_A_Subset         (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean) is
    begin
        Result := Is_A_Subset(Left,Right);
    end Is_A_Subset;

    procedure Is_A_Proper_Subset (Left        : in Set;
                                  Right       : in Set;
                                  Result      : out Boolean) is
    begin
        Result := Is_A_Proper_Subset(Left,Right);
    end Is_A_Proper_Subset;

--  end of modification

    function Is_Equal (Left  : in Set;
                       Right : in Set) return Boolean is
```

```
    Right_Index : Natural;
begin
    if Left.The_Back /= Right.The_Back then
        return False;
    else
        for Left_Index in 1 .. Left.The_Back loop
            Right_Index := Right.The_Back;
            while Right_Index > 0 loop
                if Left.The_Items(Left_Index) =
                   Right.The_Items(Right_Index) then
                    exit;
                else
                    Right_Index := Right_Index - 1;
                end if;
            end loop;
            if Right_Index = 0 then
                return False;
            end if;
        end loop;
        return True;
    end if;
end Is_Equal;

function Extent_Of (The_Set : in Set) return Natural is
begin
    return The_Set.The_Back;
end Extent_Of;

function Is_Empty (The_Set : in Set) return Boolean is
begin
    return (The_Set.The_Back = 0);
end Is_Empty;

function Is_A_Member (The_Item   : in Item;
                      Of_The_Set : in Set) return Boolean is
begin
    for Index in 1 .. Of_The_Set.The_Back loop
        if Of_The_Set.The_Items(Index) = The_Item then
            return True;
        end if;
    end loop;
    return False;
end Is_A_Member;

function Is_A_Subset (Left  : in Set;
                      Right : in Set) return Boolean is
    Right_Index : Natural;
begin
    for Left_Index in 1 .. Left.The_Back loop
        Right_Index := Right.The_Back;
        while Right_Index > 0 loop
            if Left.The_Items(Left_Index) =
               Right.The_Items(Right_Index) then
                exit;
            else
                Right_Index := Right_Index - 1;
            end if;
        end loop;
        if Right_Index = 0 then
            return False;
        end if;
    end loop;
    return True;
end Is_A_Subset;

function Is_A_Proper_Subset (Left  : in Set;
                             Right : in Set) return Boolean is
    Right_Index : Natural;
begin
    for Left_Index in 1 .. Left.The_Back loop
        Right_Index := Right.The_Back;
        while Right_Index > 0 loop
            if Left.The_Items(Left_Index) =
               Right.The_Items(Right_Index) then
                exit;
            else
                Right_Index := Right_Index - 1;
            end if;
        end loop;
        if Right_Index = 0 then
            return False;
        end if;
    end loop;
    return (Left.The_Back < Right.The_Back);
end Is_A_Proper_Subset;

procedure Iterate (Over_The_Set : in Set) is
    Continue : Boolean;
begin
    for The_Iterator in 1 .. Over_The_Set.The_Back loop
        Process(Over_The_Set.The_Items(The_Iterator), Continue);
        exit when not Continue;
    end loop;
end Iterate;

end Set_Simple_Sequential_Bounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Set_Simple_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Set : Set
    OUTPUT
      From_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

END
IMPLEMENTATION ADA Set_Simple_Sequential_Bounded_Managed_Iterator
END
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Set_Simple_Sequential_Bounded_Managed_Noniterator is

    type Set(The_Size : Positive) is limited private;

    procedure Copy          (From_The_Set : in      Set;
                             To_The_Set   : in out Set);
    procedure Clear         (The_Set      : in out Set);
    procedure Add           (The_Item     : in      Item;
                             To_The_Set   : in out Set);
    procedure Remove        (The_Item     : in      Item;
                             From_The_Set : in out Set);
    procedure Union         (Of_The_Set   : in      Set;
                             And_The_Set  : in      Set;
                             To_The_Set   : in out Set);
    procedure Intersection  (Of_The_Set   : in      Set;
                             And_The_Set  : in      Set;
                             To_The_Set   : in out Set);
    procedure Difference    (Of_The_Set   : in      Set;
                             And_The_Set  : in      Set;
                             To_The_Set   : in out Set);

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal            (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean);
    procedure Extent_Of           (The_Set    : in Set;
                                   Result     : out Natural);
    procedure Is_Empty            (The_Set    : in Set;
                                   Result     : out Boolean);
    procedure Is_A_Member         (The_Item   : in Item;
                                   Of_The_Set : in Set;
                                   Result     : out Boolean);
    procedure Is_A_Subset         (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean);
    procedure Is_A_Proper_Subset  (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean);

--  end of modification

    function Is_Equal           (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Extent_Of          (The_Set    : in Set)  return Natural;
    function Is_Empty           (The_Set    : in Set)  return Boolean;
    function Is_A_Member        (The_Item   : in Item;
                                 Of_The_Set : in Set)  return Boolean;
    function Is_A_Subset        (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Is_A_Proper_Subset (Left       : in Set;
                                 Right      : in Set)  return Boolean;

    Overflow           : exception;
    Item_Is_In_Set     : exception;
    Item_Is_Not_In_Set : exception;

private
    type Items is array(Positive range <>) of Item;
    type Set(The_Size : Positive) is
        record
            The_Back  : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Set_Simple_Sequential_Bounded_Managed_Noniterator;
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Set_Simple_Sequential_Bounded_Managed_Noniterator is

    procedure Copy (From_The_Set : in     Set;
                    To_The_Set   : in out Set) is
    begin
        if From_The_Set.The_Back > To_The_Set.The_Size then
            raise Overflow;
        else
            To_The_Set.The_Items(1 .. From_The_Set.The_Back) :=
              From_The_Set.The_Items(1 .. From_The_Set.The_Back);
            To_The_Set.The_Back := From_The_Set.The_Back;
        end if;
    end Copy;

    procedure Clear (The_Set : in out Set) is
    begin
        The_Set.The_Back := 0;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Set : in out Set) is
    begin
        for Index in 1 .. To_The_Set.The_Back loop
            if The_Item = To_The_Set.The_Items(Index) then
                raise Item_Is_In_Set;
            end if;
        end loop;
        To_The_Set.The_Items(To_The_Set.The_Back + 1) := The_Item;
        To_The_Set.The_Back := To_The_Set.The_Back + 1;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Set : in out Set) is
    begin
        for Index in 1 .. From_The_Set.The_Back loop
            if The_Item = From_The_Set.The_Items(Index) then
                From_The_Set.The_Items(Index .. (From_The_Set.The_Back
- 1)) :=
                  From_The_Set.The_Items((Index + 1) ..
From_The_Set.The_Back);
                From_The_Set.The_Back := From_The_Set.The_Back - 1;
                return;
            end if;
        end loop;
        raise Item_Is_Not_In_Set;
    end Remove;

    procedure Union (Of_The_Set : in     Set;
                     And_The_Set: in     Set;
                     To_The_Set : in out Set) is
        To_Index : Natural;
        To_Back  : Natural;
    begin
        To_The_Set.The_Items(1 .. Of_The_Set.The_Back) :=
          Of_The_Set.The_Items(1 .. Of_The_Set.The_Back);
        To_The_Set.The_Back := Of_The_Set.The_Back;
        To_Back := To_The_Set.The_Back;
        for And_Index in 1 .. And_The_Set.The_Back loop
            To_Index := To_Back;
            while To_Index > 0 loop
                if To_The_Set.The_Items(To_Index) =
                   And_The_Set.The_Items(And_Index) then
                    exit;
                else
                    To_Index := To_Index - 1;
                end if;
            end loop;
            if To_Index = 0 then
                To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
                  And_The_Set.The_Items(And_Index);
                To_The_Set.The_Back := To_The_Set.The_Back + 1;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Set  : in     Set;
                            And_The_Set : in     Set;
                            To_The_Set  : in out Set) is
        And_Index : Natural;
    begin
        To_The_Set.The_Back := 0;
        for Of_Index in 1 .. Of_The_Set.The_Back loop
            And_Index := And_The_Set.The_Back;
            while And_Index > 0 loop
                if Of_The_Set.The_Items(Of_Index) =
                   And_The_Set.The_Items(And_Index) then
                    To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
                        Of_The_Set.The_Items(Of_Index);
                    To_The_Set.The_Back := To_The_Set.The_Back + 1;
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Set  : in     Set;
                          And_The_Set : in     Set;
                          To_The_Set  : in out Set) is
        And_Index : Natural;
    begin
        To_The_Set.The_Back := 0;
        for Of_Index in 1 .. Of_The_Set.The_Back loop
            And_Index := And_The_Set.The_Back;
            while And_Index > 0 loop
                if Of_The_Set.The_Items(Of_Index) =
                   And_The_Set.The_Items(And_Index) then
                    exit;
                else
                    And_Index := And_Index - 1;
                end if;
            end loop;
            if And_Index = 0 then
                To_The_Set.The_Items(To_The_Set.The_Back + 1) :=
                    Of_The_Set.The_Items(Of_Index);
                To_The_Set.The_Back := To_The_Set.The_Back + 1;
            end if;
        end loop;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Difference;

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal            (Left      : in Set;
                                   Right     : in Set;
                                   Result    : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of           (The_Set   : in Set;
                                   Result    : out Natural) is
    begin
        Result := Extent_Of(The_Set);
    end Extent_Of;

    procedure Is_Empty            (The_Set   : in Set;
                                   Result    : out Boolean) is
    begin
        Result := Is_Empty(The_Set);
    end Is_Empty;

    procedure Is_A_Member         (The_Item   : in Item;
                                   Of_The_Set : in Set;
                                   Result     : out Boolean) is
    begin
        Result := Is_A_Member(The_Item,Of_The_Set);
    end Is_A_Member;

    procedure Is_A_Subset         (Left      : in Set;
                                   Right     : in Set;
                                   Result    : out Boolean) is
    begin
        Result := Is_A_Subset(Left,Right);
    end Is_A_Subset;

    procedure Is_A_Proper_Subset (Left      : in Set;
                                  Right     : in Set;
                                  Result    : out Boolean) is
    begin
        Result := Is_A_Proper_Subset(Left,Right);
    end Is_A_Proper_Subset;

--  end of modification

    function Is_Equal (Left  : in Set;
                       Right : in Set) return Boolean is
```

```
    Right_Index : Natural;
begin
    if Left.The_Back /= Right.The_Back then
        return False;
    else
        for Left_Index in 1 .. Left.The_Back loop
            Right_Index := Right.The_Back;
            while Right_Index > 0 loop
                if Left.The_Items(Left_Index) =
                   Right.The_Items(Right_Index) then
                    exit;
                else
                    Right_Index := Right_Index - 1;
                end if;
            end loop;
            if Right_Index = 0 then
                return False;
            end if;
        end loop;
        return True;
    end if;
end Is_Equal;

function Extent_Of (The_Set : in Set) return Natural is
begin
    return The_Set.The_Back;
end Extent_Of;

function Is_Empty (The_Set : in Set) return Boolean is
begin
    return (The_Set.The_Back = 0);
end Is_Empty;

function Is_A_Member (The_Item   : in Item;
                      Of_The_Set : in Set) return Boolean is
begin
    for Index in 1 .. Of_The_Set.The_Back loop
        if Of_The_Set.The_Items(Index) = The_Item then
            return True;
        end if;
    end loop;
    return False;
end Is_A_Member;

    function Is_A_Subset (Left  : in Set;
                          Right : in Set) return Boolean is
        Right_Index : Natural;
    begin
        for Left_Index in 1 .. Left.The_Back loop
            Right_Index := Right.The_Back;
            while Right_Index > 0 loop
                if Left.The_Items(Left_Index) =
                   Right.The_Items(Right_Index) then
                    exit;
                else
                    Right_Index := Right_Index - 1;
                end if;
            end loop;
            if Right_Index = 0 then
                return False;
            end if;
        end loop;
        return True;
    end Is_A_Subset;

    function Is_A_Proper_Subset (Left  : in Set;
                                 Right : in Set) return Boolean is
        Right_Index : Natural;
    begin
        for Left_Index in 1 .. Left.The_Back loop
            Right_Index := Right.The_Back;
            while Right_Index > 0 loop
                if Left.The_Items(Left_Index) =
                   Right.The_Items(Right_Index) then
                    exit;
                else
                    Right_Index := Right_Index - 1;
                end if;
            end loop;
            if Right_Index = 0 then
                return False;
            end if;
        end loop;
        return (Left.The_Back < Right.The_Back);
    end Is_A_Proper_Subset;

end Set_Simple_Sequential_Bounded_Managed_Noniterator;
```

SET SIMPLE SEQUENTIAL BOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Set_Simple_Sequential_Bounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Set : Set
    OUTPUT
      From_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

END
IMPLEMENTATION ADA Set_Simple_Sequential_Bounded_Managed_Noniterator
END
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Set_Simple_Sequential_Unbounded_Managed_Iterator is

    type Set is limited private;

    procedure Copy         (From_The_Set : in     Set;
                            To_The_Set   : in out Set);
    procedure Clear        (The_Set      : in out Set);
    procedure Add          (The_Item     : in     Item;
                            To_The_Set   : in out Set);
    procedure Remove       (The_Item     : in     Item;
                            From_The_Set : in out Set);
    procedure Union        (Of_The_Set   : in     Set;
                            And_The_Set  : in     Set;
                            To_The_Set   : in out Set);
    procedure Intersection (Of_The_Set   : in     Set;
                            And_The_Set  : in     Set;
                            To_The_Set   : in out Set);
    procedure Difference   (Of_The_Set   : in     Set;
                            And_The_Set  : in     Set;
                            To_The_Set   : in out Set);

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal           (Left       : in Set;
                                  Right      : in Set;
                                  Result     : out Boolean);
    procedure Extent_Of          (The_Set    : in Set;
                                  Result     : out Natural);
    procedure Is_Empty           (The_Set    : in Set;
                                  Result     : out Boolean);
    procedure Is_A_Member        (The_Item   : in Item;
                                  Of_The_Set : in Set;
                                  Result     : out Boolean);
    procedure Is_A_Subset        (Left       : in Set;
                                  Right      : in Set;
                                  Result     : out Boolean);
    procedure Is_A_Proper_Subset (Left       : in Set;
                                  Right      : in Set;
                                  Result     : out Boolean);

--  end of modification

    function Is_Equal           (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Extent_Of          (The_Set    : in Set)  return Natural;
    function Is_Empty           (The_Set    : in Set)  return Boolean;
    function Is_A_Member        (The_Item   : in Item;
                                 Of_The_Set : in Set)  return Boolean;
    function Is_A_Subset        (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Is_A_Proper_Subset (Left       : in Set;
                                 Right      : in Set)  return Boolean;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Set : in Set);

    Overflow           : exception;
    Item_Is_In_Set     : exception;
    Item_Is_Not_In_Set : exception;

private
    type Node;
    type Set is access Node;
end Set_Simple_Sequential_Unbounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body Set_Simple_Sequential_Unbounded_Managed_Iterator is

    type Node is
        record
            The_Item : Item;
            Next     : Set;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Set) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Set is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                  (Item        => Node,
                                   Pointer     => Set,
                                   Free        => Free,
                                   Set_Pointer => Set_Next,
                                   Pointer_Of  => Next_Of);

    procedure Copy (From_The_Set : in     Set;
                    To_The_Set   : in out Set) is
        From_Index : Set := From_The_Set;
        To_Index   : Set;
    begin
        Node_Manager.Free(To_The_Set);
        if From_The_Set /= null then
            To_The_Set := Node_Manager.New_Item;
            To_The_Set.The_Item := From_Index.The_Item;
            To_Index := To_The_Set;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index := To_Index.Next;
                To_Index.The_Item := From_Index.The_Item;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Set : in out Set) is
    begin
        Node_Manager.Free(The_Set);
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Set : in out Set) is
        Temporary_Node : Set;
        Index          : Set := To_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                raise Item_Is_In_Set;
            else
                Index := Index.Next;
            end if;
        end loop;
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.Next := To_The_Set;
        To_The_Set := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Set : in out Set) is
        Previous : Set;
        Index    : Set := From_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Previous = null then
                    From_The_Set := From_The_Set.Next;
                else
                    Previous.Next := Index.Next;
                end if;
                Index.Next := null;
                Node_Manager.Free(Index);
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Set;
    end Remove;

    procedure Union (Of_The_Set : in     Set;
                     And_The_Set: in     Set;
                     To_The_Set : in out Set) is
        From_Index     : Set := Of_The_Set;
        To_Index       : Set;
        To_Top         : Set;
        Temporary_Node : Set;
    begin
        Node_Manager.Free(To_The_Set);
        while From_Index /= null loop
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := From_Index.The_Item;
            Temporary_Node.Next := To_The_Set;
            To_The_Set := Temporary_Node;
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Set;
        To_Top := To_The_Set;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                Temporary_Node := Node_Manager.New_Item;
                Temporary_Node.The_Item := From_Index.The_Item;
                Temporary_Node.Next := To_The_Set;
                To_The_Set := Temporary_Node;
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Set  : in     Set;
                            And_The_Set : in     Set;
                            To_The_Set  : in out Set) is
        Of_Index       : Set := Of_The_Set;
        And_Index      : Set;
        Temporary_Node : Set;
    begin
        Node_Manager.Free(To_The_Set);
        while Of_Index /= null loop
            And_Index := And_The_Set;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    Temporary_Node := Node_Manager.New_Item;
                    Temporary_Node.The_Item := Of_Index.The_Item;
                    Temporary_Node.Next := To_The_Set;
                    To_The_Set := Temporary_Node;
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Set  : in     Set;
                          And_The_Set : in     Set;
                          To_The_Set  : in out Set) is
        Of_Index       : Set := Of_The_Set;
        And_Index      : Set;
        Temporary_Node : Set;
    begin
        Node_Manager.Free(To_The_Set);
        while Of_Index /= null loop
            And_Index := And_The_Set;
```

```
        while And_Index /= null loop
            if Of_Index.The_Item = And_Index.The_Item then
                exit;
            else
                And_Index := And_Index.Next;
            end if;
        end loop;
        if And_Index = null then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := Of_Index.The_Item;
            Temporary_Node.Next := To_The_Set;
            To_The_Set := Temporary_Node;
        end if;
        Of_Index := Of_Index.Next;
    end loop;
exception
    when Storage_Error =>
        raise Overflow;
end Difference;

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

procedure Is_Equal              (Left       : in Set;
                                 Right      : in Set;
                                 Result     : out Boolean) is
begin
    Result := Is_Equal(Left,Right);
end Is_Equal;

procedure Extent_Of             (The_Set    : in Set;
                                 Result     : out Natural) is
begin
    Result := Extent_Of(The_Set);
end Extent_Of;

procedure Is_Empty              (The_Set    : in Set;
                                 Result     : out Boolean) is
begin
    Result := Is_Empty(The_Set);
end Is_Empty;

procedure Is_A_Member           (The_Item   : in Item;
                                 Of_The_Set : in Set;
                                 Result     : out Boolean) is
begin
    Result := Is_A_Member(The_Item,Of_The_Set);
end Is_A_Member;

procedure Is_A_Subset           (Left       : in Set;
                                 Right      : in Set;
                                 Result     : out Boolean) is
begin
    Result := Is_A_Subset(Left,Right);
end Is_A_Subset;

procedure Is_A_Proper_Subset (Left       : in Set;
                                 Right      : in Set;
                                 Result     : out Boolean) is
begin
    Result := Is_A_Proper_Subset(Left,Right);
end Is_A_Proper_Subset;

--  end of modification

function Is_Equal (Left  : in Set;
                   Right : in Set) return Boolean is
    Left_Count  : Natural := 0;
    Right_Count : Natural := 0;
    Left_Index  : Set     := Left;
    Right_Index : Set;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        if Right_Index = null then
            return False;
        else
            Left_Count := Left_Count + 1;
            Left_Index := Left_Index.Next;
        end if;
    end loop;
    Right_Index := Right;
    while Right_Index /= null loop
        Right_Count := Right_Count + 1;
        Right_Index := Right_Index.Next;
    end loop;
    return (Left_Count = Right_Count);
end Is_Equal;

function Extent_Of (The_Set : in Set) return Natural is
    Count : Natural := 0;
    Index : Set     := The_Set;
begin
    while Index /= null loop
        Count := Count + 1;
        Index := Index.Next;
    end loop;
    return Count;
end Extent_Of;

function Is_Empty (The_Set : in Set) return Boolean is
begin
    return (The_Set = null);
end Is_Empty;

function Is_A_Member (The_Item   : in Item;
                      Of_The_Set : in Set) return Boolean is
    Index : Set := Of_The_Set;
begin
    while Index /= null loop
        if The_Item = Index.The_Item then
            return True;
        end if;
        Index := Index.Next;
    end loop;
    return False;
end Is_A_Member;

function Is_A_Subset (Left  : in Set;
                      Right : in Set) return Boolean is
    Left_Index  : Set := Left;
    Right_Index : Set;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        if Right_Index = null then
            return False;
        else
            Left_Index := Left_Index.Next;
        end if;
    end loop;
    return True;
end Is_A_Subset;

function Is_A_Proper_Subset (Left  : in Set;
                             Right : in Set) return Boolean is
    Left_Count  : Natural := 0;
    Right_Count : Natural := 0;
    Left_Index  : Set     := Left;
    Right_Index : Set;
begin
    while Left_Index /= null loop
        Right_Index := Right;
        while Right_Index /= null loop
            if Left_Index.The_Item = Right_Index.The_Item then
                exit;
            else
                Right_Index := Right_Index.Next;
            end if;
        end loop;
        if Right_Index = null then
            return False;
        else
            Left_Count := Left_Count + 1;
            Left_Index := Left_Index.Next;
        end if;
    end loop;
    Right_Index := Right;
    while Right_Index /= null loop
        Right_Count := Right_Count + 1;
        Right_Index := Right_Index.Next;
    end loop;
    return (Left_Count < Right_Count);
end Is_A_Proper_Subset;

procedure Iterate (Over_The_Set : in Set) is
    The_Iterator : Set      := Over_The_Set;
    Continue     : Boolean;
begin
    while The_Iterator /= null loop
        Process(The_Iterator.The_Item, Continue);
        exit when not Continue;
        The_Iterator := The_Iterator.Next;
    end loop;
end Iterate;

end Set_Simple_Sequential_Unbounded_Managed_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Set_Simple_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Set : Set
    OUTPUT
      From_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

END
IMPLEMENTATION ADA Set_Simple_Sequential_Unbounded_Managed_Iterator
END
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Set_Simple_Sequential_Unbounded_Managed_Noniterator is

    type Set is limited private;

    procedure Copy          (From_The_Set : in     Set;
                             To_The_Set   : in out Set);
    procedure Clear         (The_Set      : in out Set);
    procedure Add           (The_Item     : in     Item;
                             To_The_Set   : in out Set);
    procedure Remove        (The_Item     : in     Item;
                             From_The_Set : in out Set);
    procedure Union         (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);
    procedure Intersection  (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);
    procedure Difference    (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal            (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean);
    procedure Extent_Of           (The_Set    : in Set;
                                   Result     : out Natural);
    procedure Is_Empty            (The_Set    : in Set;
                                   Result     : out Boolean);
    procedure Is_A_Member         (The_Item   : in Item;
                                   Of_The_Set : in Set;
                                   Result     : out Boolean);
    procedure Is_A_Subset         (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean);
    procedure Is_A_Proper_Subset  (Left       : in Set;
                                   Right      : in Set;
                                   Result     : out Boolean);

--  end of modification

    function Is_Equal            (Left       : in Set;
                                  Right      : in Set)  return Boolean;
    function Extent_Of           (The_Set    : in Set)  return Natural;
    function Is_Empty            (The_Set    : in Set)  return Boolean;
    function Is_A_Member         (The_Item   : in Item;
                                  Of_The_Set : in Set)  return Boolean;
    function Is_A_Subset         (Left       : in Set;
                                  Right      : in Set)  return Boolean;
    function Is_A_Proper_Subset  (Left       : in Set;
                                  Right      : in Set)  return Boolean;

    Overflow            : exception;
    Item_Is_In_Set      : exception;
    Item_Is_Not_In_Set  : exception;

private
    type Node;
    type Set is access Node;
end Set_Simple_Sequential_Unbounded_Managed_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body Set_Simple_Sequential_Unbounded_Managed_Noniterator is

    type Node is
        record
            The_Item : Item;
            Next     : Set;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Set) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Set is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                  (Item        => Node,
                                   Pointer     => Set,
                                   Free        => Free,
                                   Set_Pointer => Set_Next,
                                   Pointer_Of  => Next_Of);

    procedure Copy (From_The_Set : in     Set;
                    To_The_Set   : in out Set) is
        From_Index : Set := From_The_Set;
        To_Index   : Set;
    begin
        Node_Manager.Free(To_The_Set);
        if From_The_Set /= null then
            To_The_Set := Node_Manager.New_Item;
            To_The_Set.The_Item := From_Index.The_Item;
            To_Index := To_The_Set;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index := To_Index.Next;
                To_Index.The_Item := From_Index.The_Item;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Set : in out Set) is
    begin
        Node_Manager.Free(The_Set);
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Set : in out Set) is
        Temporary_Node : Set;
        Index          : Set := To_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                raise Item_Is_In_Set;
            else
                Index := Index.Next;
            end if;
        end loop;
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.Next := To_The_Set;
        To_The_Set := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Set : in out Set) is
        Previous : Set;
        Index    : Set := From_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Previous = null then
                    From_The_Set := From_The_Set.Next;
                else
                    Previous.Next := Index.Next;
                end if;
                Index.Next := null;
                Node_Manager.Free(Index);
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Set;
    end Remove;

    procedure Union (Of_The_Set : in     Set;
                     And_The_Set: in     Set;
                     To_The_Set : in out Set) is
        From_Index     : Set := Of_The_Set;
        To_Index       : Set;
        To_Top         : Set;
        Temporary_Node : Set;
    begin
        Node_Manager.Free(To_The_Set);
        while From_Index /= null loop
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := From_Index.The_Item;
            Temporary_Node.Next := To_The_Set;
            To_The_Set := Temporary_Node;
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Set;
        To_Top := To_The_Set;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                Temporary_Node := Node_Manager.New_Item;
                Temporary_Node.The_Item := From_Index.The_Item;
                Temporary_Node.Next := To_The_Set;
                To_The_Set := Temporary_Node;
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Set  : in     Set;
                            And_The_Set : in     Set;
                            To_The_Set  : in out Set) is
        Of_Index       : Set := Of_The_Set;
        And_Index      : Set;
        Temporary_Node : Set;
    begin
        Node_Manager.Free(To_The_Set);
        while Of_Index /= null loop
            And_Index := And_The_Set;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    Temporary_Node := Node_Manager.New_Item;
                    Temporary_Node.The_Item := Of_Index.The_Item;
                    Temporary_Node.Next := To_The_Set;
                    To_The_Set := Temporary_Node;
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Set  : in     Set;
                          And_The_Set : in     Set;
                          To_The_Set  : in out Set) is
        Of_Index       : Set := Of_The_Set;
        And_Index      : Set;
        Temporary_Node : Set;
    begin
        Node_Manager.Free(To_The_Set);
        while Of_Index /= null loop
            And_Index := And_The_Set;
```

```
        while And_Index /= null loop
            if Of_Index.The_Item = And_Index.The_Item then
                exit;
            else
                And_Index := And_Index.Next;
            end if;
        end loop;
        if And_Index = null then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := Of_Index.The_Item;
            Temporary_Node.Next := To_The_Set;
            To_The_Set := Temporary_Node;
        end if;
        Of_Index := Of_Index.Next;
    end loop;
exception
    when Storage_Error =>
        raise Overflow;
end Difference;

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal              (Left       : in Set;
                                     Right      : in Set;
                                     Result     : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of             (The_Set    : in Set;
                                     Result     : out Natural) is
    begin
        Result := Extent_Of(The_Set);
    end Extent_Of;

    procedure Is_Empty              (The_Set    : in Set;
                                     Result     : out Boolean) is
    begin
        Result := Is_Empty(The_Set);
    end Is_Empty;

    procedure Is_A_Member           (The_Item   : in Item;
                                     Of_The_Set : in Set;
                                     Result     : out Boolean) is
    begin
        Result := Is_A_Member(The_Item,Of_The_Set);
    end Is_A_Member;

    procedure Is_A_Subset           (Left       : in Set;
                                     Right      : in Set;
                                     Result     : out Boolean) is
    begin
        Result := Is_A_Subset(Left,Right);
    end Is_A_Subset;

    procedure Is_A_Proper_Subset    (Left       : in Set;
                                     Right      : in Set;
                                     Result     : out Boolean) is
    begin
        Result := Is_A_Proper_Subset(Left,Right);
    end Is_A_Proper_Subset;

--  end of modification

    function Is_Equal (Left  : in Set;
                       Right : in Set) return Boolean is
        Left_Count  : Natural := 0;
        Right_Count : Natural := 0;
        Left_Index  : Set     := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Count := Left_Count + 1;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Right_Count := Right_Count + 1;
            Right_Index := Right_Index.Next;
        end loop;
        return (Left_Count = Right_Count);
    end Is_Equal;

    function Extent_Of (The_Set : in Set) return Natural is
        Count : Natural := 0;
        Index : Set     := The_Set;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Extent_Of;

    function Is_Empty (The_Set : in Set) return Boolean is
    begin
        return (The_Set = null);
    end Is_Empty;

    function Is_A_Member (The_Item   : in Item;
                          Of_The_Set : in Set) return Boolean is
        Index : Set := Of_The_Set;
    begin
        while Index /= null loop
            if The_Item = Index.The_Item then
                return True;
            end if;
            Index := Index.Next;
        end loop;
        return False;
    end Is_A_Member;

    function Is_A_Subset (Left  : in Set;
                          Right : in Set) return Boolean is
        Left_Index  : Set := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        return True;
    end Is_A_Subset;

    function Is_A_Proper_Subset (Left  : in Set;
                                 Right : in Set) return Boolean is
        Left_Count  : Natural := 0;
        Right_Count : Natural := 0;
        Left_Index  : Set     := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Count := Left_Count + 1;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Right_Count := Right_Count + 1;
            Right_Index := Right_Index.Next;
        end loop;
        return (Left_Count < Right_Count);
    end Is_A_Proper_Subset;

end Set_Simple_Sequential_Unbounded_Managed_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Set_Simple_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Set : Set
    OUTPUT
      From_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

END
IMPLEMENTATION ADA Set_Simple_Sequential_Unbounded_Managed_Noniterator
END
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Set_Simple_Sequential_Unbounded_Unmanaged_Iterator is

    type Set is limited private;

    procedure Copy          (From_The_Set : in     Set;
                             To_The_Set   : in out Set);
    procedure Clear         (The_Set      : in out Set);
    procedure Add           (The_Item     : in     Item;
                             To_The_Set   : in out Set);
    procedure Remove        (The_Item     : in     Item;
                             From_The_Set : in out Set);
    procedure Union         (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);
    procedure Intersection (Of_The_Set    : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);
    procedure Difference    (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal          (Left       : in Set;
                                 Right      : in Set;
                                 Result     : out Boolean);
    procedure Extent_Of         (The_Set    : in Set;
                                 Result     : out Natural);
    procedure Is_Empty          (The_Set    : in Set;
                                 Result     : out Boolean);
    procedure Is_A_Member       (The_Item   : in Item;
                                 Of_The_Set : in Set;
                                 Result     : out Boolean);
    procedure Is_A_Subset       (Left       : in Set;
                                 Right      : in Set;
                                 Result     : out Boolean);
    procedure Is_A_Proper_Subset (Left      : in Set;
                                 Right      : in Set;
                                 Result     : out Boolean);

--  end of modification

    function Is_Equal           (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Extent_Of          (The_Set    : in Set)  return Natural;
    function Is_Empty           (The_Set    : in Set)  return Boolean;
    function Is_A_Member        (The_Item   : in Item;
                                 Of_The_Set : in Set)  return Boolean;
    function Is_A_Subset        (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Is_A_Proper_Subset (Left       : in Set;
                                 Right      : in Set)  return Boolean;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_Set : in Set);

    Overflow           : exception;
    Item_Is_In_Set     : exception;
    Item_Is_Not_In_Set : exception;

private
    type Node;
    type Set is access Node;
end Set_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Set_Simple_Sequential_Unbounded_Unmanaged_Iterator is

    type Node is
        record
            The_Item : Item;
            Next     : Set;
        end record;

    procedure Copy (From_The_Set : in     Set;
                    To_The_Set   : in out Set) is
        From_Index : Set := From_The_Set;
        To_Index   : Set;
    begin
        if From_The_Set = null then
            To_The_Set := null;
        else
            To_The_Set := new Node'(The_Item => From_Index.The_Item,
                                    Next     => null);
            To_Index := To_The_Set;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Set : in out Set) is
    begin
        The_Set := null;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Set : in out Set) is
        Index : Set := To_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                raise Item_Is_In_Set;
            else
                Index := Index.Next;
            end if;
        end loop;
        To_The_Set := new Node'(The_Item => The_Item,
                                Next     => To_The_Set);
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Set : in out Set) is
        Previous : Set;
        Index    : Set := From_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Previous = null then
                    From_The_Set := From_The_Set.Next;
                else
                    Previous.Next := Index.Next;
                end if;
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Set;
    end Remove;

    procedure Union (Of_The_Set : in     Set;
                     And_The_Set: in     Set;
                     To_The_Set : in out Set) is
        From_Index : Set := Of_The_Set;
        To_Index   : Set;
        To_Top     : Set;
    begin
        To_The_Set := null;
        while From_Index /= null loop
            To_The_Set := new Node'(The_Item => From_Index.The_Item,
                                    Next     => To_The_Set);
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Set;
        To_Top := To_The_Set;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                To_The_Set := new Node'(The_Item =>
From_Index.The_Item,
                                        Next     => To_The_Set);
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Set  : in     Set;
                            And_The_Set : in     Set;
                            To_The_Set  : in out Set) is
        Of_Index  : Set := Of_The_Set;
        And_Index : Set;
    begin
        To_The_Set := null;
        while Of_Index /= null loop
            And_Index := And_The_Set;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    To_The_Set := new Node'(The_Item =>
Of_Index.The_Item,
                                            Next     => To_The_Set);
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Set  : in     Set;
                          And_The_Set : in     Set;
                          To_The_Set  : in out Set) is
        Of_Index  : Set := Of_The_Set;
        And_Index : Set;
    begin
        To_The_Set:= null;
        while Of_Index /= null loop
            And_Index := And_The_Set;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            if And_Index = null then
                To_The_Set := new Node'(The_Item => Of_Index.The_Item,
                                        Next     => To_The_Set);
            end if;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Difference;

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal          (Left     : in Set;
                                 Right    : in Set;
                                 Result   : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of         (The_Set  : in Set;
                                 Result   : out Natural) is
    begin
        Result := Extent_Of(The_Set);
```

```
    end Extent_Of;

    procedure Is_Empty             (The_Set     : in Set;
                                     Result      : out Boolean) is
    begin
        Result := Is_Empty(The_Set);
    end Is_Empty;

    procedure Is_A_Member          (The_Item    : in Item;
                                     Of_The_Set  : in Set;
                                     Result      : out Boolean) is
    begin
        Result := Is_A_Member(The_Item,Of_The_Set);
    end Is_A_Member;

    procedure Is_A_Subset          (Left        : in Set;
                                     Right       : in Set;
                                     Result      : out Boolean) is
    begin
        Result := Is_A_Subset(Left,Right);
    end Is_A_Subset;

    procedure Is_A_Proper_Subset (Left         : in Set;
                                     Right       : in Set;
                                     Result      : out Boolean) is
    begin
        Result := Is_A_Proper_Subset(Left,Right);
    end Is_A_Proper_Subset;

--  end of modification

    function Is_Equal (Left  : in Set;
                       Right : in Set) return Boolean is
        Left_Count  : Natural := 0;
        Right_Count : Natural := 0;
        Left_Index  : Set     := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Count := Left_Count + 1;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Right_Count := Right_Count + 1;
            Right_Index := Right_Index.Next;
        end loop;
        return (Left_Count = Right_Count);
    end Is_Equal;

    function Extent_Of (The_Set : in Set) return Natural is
        Count : Natural := 0;
        Index : Set     := The_Set;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Extent_Of;

    function Is_Empty (The_Set : in Set) return Boolean is
    begin
        return (The_Set = null);
    end Is_Empty;

    function Is_A_Member (The_Item   : in Item;
                          Of_The_Set : in Set) return Boolean is
        Index : Set := Of_The_Set;
    begin
        while Index /= null loop
            if The_Item = Index.The_Item then
                return True;
            end if;
            Index := Index.Next;
        end loop;
        return False;
    end Is_A_Member;

    function Is_A_Subset (Left  : in Set;
                          Right : in Set) return Boolean is
        Left_Index  : Set := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        return True;
    end Is_A_Subset;

    function Is_A_Proper_Subset (Left  : in Set;
                                 Right : in Set) return Boolean is
        Left_Count  : Natural := 0;
        Right_Count : Natural := 0;
        Left_Index  : Set     := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Count := Left_Count + 1;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Right_Count := Right_Count + 1;
            Right_Index := Right_Index.Next;
        end loop;
        return (Left_Count < Right_Count);
    end Is_A_Proper_Subset;

    procedure Iterate (Over_The_Set : in Set) is
        The_Iterator : Set      := Over_The_Set;
        Continue     : Boolean;
    begin
        while The_Iterator /= null loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Set_Simple_Sequential_Unbounded_Unmanaged_Iterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Set_Simple_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Set : Set
    OUTPUT
      From_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
    INPUT
      Over_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

END
IMPLEMENTATION ADA Set_Simple_Sequential_Unbounded_Unmanaged_Iterator
END
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator is

    type Set is limited private;

    procedure Copy          (From_The_Set : in     Set;
                             To_The_Set   : in out Set);
    procedure Clear         (The_Set      : in out Set);
    procedure Add           (The_Item     : in     Item;
                             To_The_Set   : in out Set);
    procedure Remove        (The_Item     : in     Item;
                             From_The_Set : in out Set);
    procedure Union         (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);
    procedure Intersection  (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);
    procedure Difference    (Of_The_Set   : in     Set;
                             And_The_Set  : in     Set;
                             To_The_Set   : in out Set);

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal             (Left        : in Set;
                                    Right       : in Set;
                                    Result      : out Boolean);
    procedure Extent_Of            (The_Set     : in Set;
                                    Result      : out Natural);
    procedure Is_Empty             (The_Set     : in Set;
                                    Result      : out Boolean);
    procedure Is_A_Member          (The_Item    : in Item;
                                    Of_The_Set  : in Set;
                                    Result      : out Boolean);
    procedure Is_A_Subset          (Left        : in Set;
                                    Right       : in Set;
                                    Result      : out Boolean);
    procedure Is_A_Proper_Subset   (Left        : in Set;
                                    Right       : in Set;
                                    Result      : out Boolean);

--  end of modification

    function Is_Equal           (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Extent_Of          (The_Set    : in Set)  return Natural;
    function Is_Empty           (The_Set    : in Set)  return Boolean;
    function Is_A_Member        (The_Item   : in Item;
                                 Of_The_Set : in Set)  return Boolean;
    function Is_A_Subset        (Left       : in Set;
                                 Right      : in Set)  return Boolean;
    function Is_A_Proper_Subset (Left       : in Set;
                                 Right      : in Set)  return Boolean;

    Overflow             : exception;
    Item_Is_In_Set       : exception;
    Item_Is_Not_In_Set   : exception;

private
    type Node;
    type Set is access Node;
end Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator is

    type Node is
        record
            The_Item : Item;
            Next     : Set;
        end record;

    procedure Copy (From_The_Set : in     Set;
                    To_The_Set   : in out Set) is
        From_Index : Set := From_The_Set;
        To_Index   : Set;
    begin
        if From_The_Set = null then
            To_The_Set := null;
        else
            To_The_Set := new Node'(The_Item => From_Index.The_Item,
                                    Next     => null);
            To_Index := To_The_Set;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     => null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Set : in out Set) is
    begin
        The_Set := null;
    end Clear;

    procedure Add (The_Item   : in     Item;
                   To_The_Set : in out Set) is
        Index : Set := To_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                raise Item_Is_In_Set;
            else
                Index := Index.Next;
            end if;
        end loop;
        To_The_Set := new Node'(The_Item => The_Item,
                                Next     => To_The_Set);
    exception
        when Storage_Error =>
            raise Overflow;
    end Add;

    procedure Remove (The_Item     : in     Item;
                      From_The_Set : in out Set) is
        Previous : Set;
        Index    : Set := From_The_Set;
    begin
        while Index /= null loop
            if Index.The_Item = The_Item then
                if Previous = null then
                    From_The_Set := From_The_Set.Next;
                else
                    Previous.Next := Index.Next;
                end if;
                return;
            else
                Previous := Index;
                Index := Index.Next;
            end if;
        end loop;
        raise Item_Is_Not_In_Set;
    end Remove;

    procedure Union (Of_The_Set : in     Set;
                     And_The_Set: in     Set;
                     To_The_Set : in out Set) is
        From_Index : Set := Of_The_Set;
        To_Index   : Set;
        To_Top     : Set;
    begin
        To_The_Set := null;
        while From_Index /= null loop
            To_The_Set := new Node'(The_Item => From_Index.The_Item,
                                    Next     => To_The_Set);
            From_Index := From_Index.Next;
        end loop;
        From_Index := And_The_Set;
        To_Top := To_The_Set;
        while From_Index /= null loop
            To_Index := To_Top;
            while To_Index /= null loop
                if From_Index.The_Item = To_Index.The_Item then
                    exit;
                else
                    To_Index := To_Index.Next;
                end if;
            end loop;
            if To_Index = null then
                To_The_Set := new Node'(The_Item =>
From_Index.The_Item,
                                        Next     => To_The_Set);
            end if;
            From_Index := From_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Union;

    procedure Intersection (Of_The_Set  : in     Set;
                            And_The_Set : in     Set;
                            To_The_Set  : in out Set) is
        Of_Index  : Set := Of_The_Set;
        And_Index : Set;
    begin
        To_The_Set := null;
        while Of_Index /= null loop
            And_Index := And_The_Set;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    To_The_Set := new Node'(The_Item =>
Of_Index.The_Item,
                                            Next     => To_The_Set);
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Intersection;

    procedure Difference (Of_The_Set  : in     Set;
                          And_The_Set : in     Set;
                          To_The_Set  : in out Set) is
        Of_Index  : Set := Of_The_Set;
        And_Index : Set;
    begin
        To_The_Set:= null;
        while Of_Index /= null loop
            And_Index := And_The_Set;
            while And_Index /= null loop
                if Of_Index.The_Item = And_Index.The_Item then
                    exit;
                else
                    And_Index := And_Index.Next;
                end if;
            end loop;
            if And_Index = null then
                To_The_Set := new Node'(The_Item => Of_Index.The_Item,
                                        Next     => To_The_Set);
            end if;
            Of_Index := Of_Index.Next;
        end loop;
    exception
        when Storage_Error =>
            raise Overflow;
    end Difference;

--  modified by Tuan Nguyen
--  20 Aug 95
--  replacing functions with procedures

    procedure Is_Equal          (Left      : in Set;
                                 Right     : in Set;
                                 Result    : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Extent_Of         (The_Set   : in Set;
                                 Result    : out Natural) is
    begin
        Result := Extent_Of(The_Set);
```

```
    end Extent_Of;

    procedure Is_Empty             (The_Set     : in Set;
                                    Result      : out Boolean) is
    begin
        Result := Is_Empty(The_Set);
    end Is_Empty;

    procedure Is_A_Member          (The_Item    : in Item;
                                    Of_The_Set  : in Set;
                                    Result      : out Boolean) is
    begin
        Result := Is_A_Member(The_Item,Of_The_Set);
    end Is_A_Member;

    procedure Is_A_Subset          (Left        : in Set;
                                    Right       : in Set;
                                    Result      : out Boolean) is
    begin
        Result := Is_A_Subset(Left,Right);
    end Is_A_Subset;

    procedure Is_A_Proper_Subset   (Left        : in Set;
                                    Right       : in Set;
                                    Result      : out Boolean) is
    begin
        Result := Is_A_Proper_Subset(Left,Right);
    end Is_A_Proper_Subset;

--  end of modification

    function Is_Equal (Left  : in Set;
                       Right : in Set) return Boolean is
        Left_Count  : Natural := 0;
        Right_Count : Natural := 0;
        Left_Index  : Set     := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Count := Left_Count + 1;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Right_Count := Right_Count + 1;
            Right_Index := Right_Index.Next;
        end loop;
        return (Left_Count = Right_Count);
    end Is_Equal;

    function Extent_Of (The_Set : in Set) return Natural is
        Count : Natural := 0;
        Index : Set     := The_Set;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Extent_Of;

    function Is_Empty (The_Set : in Set) return Boolean is
    begin
        return (The_Set = null);
    end Is_Empty;

    function Is_A_Member (The_Item   : in Item;
                          Of_The_Set : in Set) return Boolean is
        Index : Set := Of_The_Set;
    begin
        while Index /= null loop
            if The_Item = Index.The_Item then
                return True;
            end if;
            Index := Index.Next;
        end loop;
        return False;
    end Is_A_Member;

    function Is_A_Subset (Left  : in Set;
                          Right : in Set) return Boolean is
        Left_Index  : Set := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        return True;
    end Is_A_Subset;

    function Is_A_Proper_Subset (Left  : in Set;
                                 Right : in Set) return Boolean is
        Left_Count  : Natural := 0;
        Right_Count : Natural := 0;
        Left_Index  : Set     := Left;
        Right_Index : Set;
    begin
        while Left_Index /= null loop
            Right_Index := Right;
            while Right_Index /= null loop
                if Left_Index.The_Item = Right_Index.The_Item then
                    exit;
                else
                    Right_Index := Right_Index.Next;
                end if;
            end loop;
            if Right_Index = null then
                return False;
            else
                Left_Count := Left_Count + 1;
                Left_Index := Left_Index.Next;
            end if;
        end loop;
        Right_Index := Right;
        while Right_Index /= null loop
            Right_Count := Right_Count + 1;
            Right_Index := Right_Index.Next;
        end loop;
        return (Left_Count < Right_Count);
    end Is_A_Proper_Subset;

end Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator;
```

SET SIMPLE SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Add
  SPECIFICATION
    INPUT
      The_Item : Item,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Remove
  SPECIFICATION
    INPUT
      The_Item : Item,
      From_The_Set : Set
    OUTPUT
      From_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Union
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Intersection
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Difference
  SPECIFICATION
    INPUT
      Of_The_Set : Set,
      And_The_Set : Set,
      To_The_Set : Set
    OUTPUT
      To_The_Set : Set
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Extent_Of
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Member
  SPECIFICATION
    INPUT
      The_Item : Item,
      Of_The_Set : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

  OPERATOR Is_A_Proper_Subset
  SPECIFICATION
    INPUT
      Left : Set,
      Right : Set
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Item_Is_In_Set, Item_Is_Not_In_Set
  END

END
IMPLEMENTATION ADA
Set_Simple_Sequential_Unbounded_Unmanaged_Noniterator
END
```

BINARY SEARCH

ADA SPECIFICATIONS

```
generic
    type Key   is limited private;
    type Item  is limited private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function Is_Equal (Left  : in Key;
                            Right : in Item) return
Boolean;
    with function Is_Less_Than (Left  : in Key;
                                Right : in Item) return
Boolean;
package Binary_Search is

-- modified by Tuan Nguyen
-- 20 Jan 95
```

```
-- adding procedures to replace functions

    procedure Location_Of (The_Key      : in Key;
                           In_The_Items : in Items;
                           Result       : out Index);

-- end of modification

    function Location_Of (The_Key      : in Key;
                          In_The_Items : in Items)
return Index;

    Item_Not_Found : exception;

end Binary_Search;
```

BINARY SEARCH

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Binary_Search is

-- modified by Tuan Nguyen
-- 20 Jan 95
-- adding procedures to replace functions

    procedure Location_Of (The_Key      : in Key;
                           In_The_Items : in Items;
                           Result       : out Index) is
    begin
        Result := Location_Of(The_Key,In_The_Items);
    end Location_Of;

-- end of modification
```

```
    function Location_Of (The_Key      : in Key;
                          In_The_Items : in Items)
return Index is
        Lower_Index : Index := In_The_Items'First;
        Upper_Index : Index := In_The_Items'Last;
        The_Index   : Index;
    begin
        while Lower_Index <= Upper_Index loop
            The_Index :=
               Index'Val((Index'Pos(Lower_Index) +
Index'Pos(Upper_Index)) / 2);
            if Is_Equal(The_Key,
In_The_Items(The_Index)) then
                return The_Index;
            elsif Is_Less_Than(The_Key,
In_The_Items(The_Index)) then
                exit when (The_Index =
In_The_Items'First);
                Upper_Index := Index'Pred(The_Index);
            else
                exit when (The_Index =
In_The_Items'Last);
                Lower_Index := Index'Succ(The_Index);
            end if;
        end loop;
        raise Item_Not_Found;
    end Location_Of;

end Binary_Search;
```

BINARY SEARCH

PSDL

```
OPERATOR Location_Of
SPECIFICATION
  GENERIC
    Key : PRIVATE_TYPE,
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    Is_Equal : FUNCTION[Left : Key, Right : Item,
RETURN : Boolean],
    Is_Less_Than : FUNCTION[Left : Key, Right : Item,
RETURN : Boolean]
```

```
  INPUT
    The_Key : Key,
    In_The_Items : Items
  OUTPUT
    Result : Index
  EXCEPTIONS
    Item_Not_Found
END

IMPLEMENTATION ADA Location_Of
END
```

BINARY INSERTION SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Binary_Insertion_Sort is

    procedure Sort (The_Items : in out Items);

end Binary_Insertion_Sort;
```

BINARY INSERTION SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Binary_Insertion_Sort is

    procedure Sort (The_Items : in out Items) is
        Temporary_Item : Item;
        Left_Index     : Index;
        Middle_Index   : Index;
        Right_Index    : Index;
    begin
        for Outer_Index in Index'Succ(The_Items'First)
.. The_Items'Last loop
            Temporary_Item := The_Items(Outer_Index);
            Left_Index := The_Items'First;
            Right_Index := Outer_Index;
            while Left_Index <= Right_Index loop
                Middle_Index :=
Index'Val((Index'Pos(Left_Index) +

Index'Pos(Right_Index)) / 2);
                if Temporary_Item <
The_Items(Middle_Index) then
                    exit when (Middle_Index =
The_Items'First);
                    Right_Index :=
Index'Pred(Middle_Index);
                else
                    exit when (Middle_Index =
Outer_Index);
                    Left_Index :=
Index'Succ(Middle_Index);
                end if;
            end loop;
            if Left_Index /= Outer_Index then
                The_Items(Index'Succ(Left_Index) ..
Outer_Index) :=
                  The_Items(Left_Index ..
Index'Pred(Outer_Index));
                The_Items(Left_Index) :=
Temporary_Item;
            end if;
        end loop;
    end Sort;

end Binary_Insertion_Sort;
```

BINARY INSERTION SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

BUBBLE SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Bubble_Sort is

    procedure Sort (The_Items : in out Items);

end Bubble_Sort;
```

BUBBLE SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Bubble_Sort is

    procedure Sort (The_Items : in out Items) is
        Temporary_Item : Item;
        Exchanges_Made : Boolean;
    begin
        for Outer_Index in Index'Succ(The_Items'First)
.. The_Items'Last loop
            Exchanges_Made := False;
            for Inner_Index in reverse Outer_Index ..
The_Items'Last loop
                if The_Items(Inner_Index) <
                  The_Items(Index'Pred(Inner_Index))
then
                    Exchanges_Made := True;
                    Temporary_Item :=
The_Items(Index'Pred(Inner_Index));
                    The_Items(Index'Pred(Inner_Index))
:=
                      The_Items(Inner_Index);
                    The_Items(Inner_Index) :=
Temporary_Item;
                end if;
            end loop;
            exit when not Exchanges_Made;
        end loop;
    end Sort;

end Bubble_Sort;
```

BUBBLE SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

HEAP SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Heap_Sort is

    procedure Sort (The_Items : in out Items);

end Heap_Sort;
```

HEAP SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--             "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Heap_Sort is

    procedure Sort (The_Items : in out Items) is

        Temporary_Item : Item;
        Left_Index     : Index;
        Right_Index    : Index;

        procedure Sift (Left_Index  : Index;
                        Right_Index : Index) is
            Temporary_Item : Item  :=
The_Items(Left_Index);
            The_Front      : Index := Left_Index;
            The_Back       : Index :=
Index'Val(Index'Pos(The_Front) * 2);
        begin
            while The_Back <= Right_Index loop
                if The_Back < Right_Index then
                    if The_Items(The_Back) <
                       The_Items(Index'Succ(The_Back))
then
                        The_Back :=
Index'Succ(The_Back);
                    end if;
                end if;
                exit when not (Temporary_Item <
The_Items(The_Back));
                The_Items(The_Front) :=
The_Items(The_Back);
                The_Front := The_Back;
                exit when (Index'Pos(The_Front) * 2 >
                           Index'Pos(The_Items'Last));
                The_Back :=
Index'Val(Index'Pos(The_Front) * 2);
            end loop;
            The_Items(The_Front) := Temporary_Item;
        end Sift;

    begin
        Left_Index :=
          Index'Val(((Index'Pos(The_Items'Last) -
                      Index'Pos(The_Items'First) + 1) /
2) + 1);
        Right_Index := The_Items'Last;
        while Left_Index > The_Items'First loop
            Left_Index := Index'Pred(Left_Index);
            Sift(Left_Index, Right_Index);
        end loop;
        while Right_Index > The_Items'First loop
            Temporary_Item :=
The_Items(The_Items'First);
            The_Items(The_Items'First) :=
The_Items(Right_Index);
            The_Items(Right_Index) := Temporary_Item;
            Right_Index := Index'Pred(Right_Index);
            Sift(Left_Index, Right_Index);
        end loop;
    end Sort;

end Heap_Sort;
```

HEAP SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

NATURAL MERGE SORT

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type File is limited private;
    with procedure Open_For_Reading (The_File : in out
File);
    with procedure Open_For_Writing (The_File : in out
File);
    with procedure Get              (The_File : in out
File;
                                     The_Item : out
Item);
    with procedure Put              (The_File : in out
File;
                                     The_Item : in
Item);
    with procedure Close            (The_File : in out
File);
    with function Next_Item        (The_File : in File)
return Item;
    with function "<"              (Left     : in Item;
                                     Right    : in Item)
return Boolean;
    with function Is_End_Of_File (The_File : in File)
return Boolean;
package Natural_Merge_Sort is

    procedure Sort (The_File         : in out File;
                    Temporary_File_1 : in out File;
                    Temporary_File_2 : in out File);

    File_Is_Empty : exception;

end Natural_Merge_Sort;
```

NATURAL MERGE SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Natural_Merge_Sort is

    procedure Sort (The_File         : in out File;
                    Temporary_File_1 : in out File;
                    Temporary_File_2 : in out File) is

        Number_Of_Runs : Natural;

        procedure Copy (From_The_File : in out File;
                        To_The_File   : in out File;
                        End_Of_Run    : out    Boolean)
is
            Temporary_Item : Item;
        begin
            Get(From_The_File, Temporary_Item);
            Put(To_The_File, Temporary_Item);
            if Is_End_Of_File(From_The_File) then
                End_Of_Run := True;
            else
                End_Of_Run := (Next_Item(From_The_File)
< Temporary_Item);
            end if;
        end Copy;

        procedure Copy_Run (From_The_File : in out
File;
                            To_The_File   : in out
File) is
            End_Of_Run : Boolean;
        begin
            loop
                Copy(From_The_File, To_The_File,
End_Of_Run);
                exit when End_Of_Run;
            end loop;
        end Copy_Run;

        procedure Merge_Run (From_The_File : in out
File;
                             And_The_File  : in out
File;
                             To_The_File   : in out
File) is
            End_Of_Run : Boolean;
        begin
            loop
                if not (Next_Item(And_The_File) <
                        Next_Item(From_The_File)) then
                    Copy(From_The_File, To_The_File,
End_Of_Run);
                    if End_Of_Run then
                        Copy_Run(And_The_File,
To_The_File);
                        exit;
                    end if;
                else
                    Copy(And_The_File, To_The_File,
End_Of_Run);
                    if End_Of_Run then
                        Copy_Run(From_The_File,
To_The_File);
                        exit;
                    end if;
                end if;
            end loop;
        end Merge_Run;

    begin
        loop
            Open_For_Reading(The_File);
            if Is_End_Of_File(The_File) then
                Close(The_File);
                Close(Temporary_File_1);
                Close(Temporary_File_2);
                raise File_Is_Empty;
            else
                Open_For_Writing(Temporary_File_1);
                Open_For_Writing(Temporary_File_2);
                loop
                    Copy_Run(The_File, To_The_File =>
Temporary_File_1);
                    if not Is_End_Of_File(The_File)
then
                        Copy_Run(The_File, To_The_File
=> Temporary_File_2);
                    end if;
                    exit when Is_End_Of_File(The_File);
                end loop;
                Open_For_Writing(The_File);
                Open_For_Reading(Temporary_File_1);
                Open_For_Reading(Temporary_File_2);
                Number_Of_Runs := 0;
                while (not
Is_End_Of_File(Temporary_File_1)) and
                      (not
Is_End_Of_File(Temporary_File_2)) loop
                    Merge_Run(Temporary_File_1,
Temporary_File_2,
                              To_The_File => The_File);
                    Number_Of_Runs := Number_Of_Runs +
1;
                end loop;
                while not
Is_End_Of_File(Temporary_File_1) loop
                    Copy_Run(Temporary_File_1,
To_The_File => The_File);
                    Number_Of_Runs := Number_Of_Runs +
1;
                end loop;
                while not
Is_End_Of_File(Temporary_File_2) loop
                    Copy_Run(Temporary_File_2,
To_The_File => The_File);
                    Number_Of_Runs := Number_Of_Runs +
1;
                end loop;
                exit when (Number_Of_Runs = 1);
            end if;
        end loop;
        Close(The_File);
        Close(Temporary_File_1);
        Close(Temporary_File_2);
    end Sort;

end Natural_Merge_Sort;
```

NATURAL MERGE SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    File : PRIVATE_TYPE,
    Open_For_Reading : PROCEDURE[The_File : in_out[t :
File]],
    Open_For_Writing : PROCEDURE[The_File : in_out[t :
File]],
    Get : PROCEDURE[The_File : in_out[t : File],
The_Item : out[t : Item]],
    Put : PROCEDURE[The_File : in_out[t : File],
The_Item : in[t : Item]],
    Close : PROCEDURE[The_File : in_out[t : File]],
    Next_Item : FUNCTION[The_File : File, RETURN :
Item],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean],
    Is_End_Of_File : FUNCTION[The_File : File, RETURN :
Boolean]
  INPUT
    The_File : File,
    Temporary_File_1 : File,
    Temporary_File_2 : File
  OUTPUT
    The_File : File,
    Temporary_File_1 : File,
    Temporary_File_2 : File
  EXCEPTIONS
    File_Is_Empty
END

IMPLEMENTATION ADA Sort
END
```

ORDERED SEQUENTIAL SEARCH

ADA SPECIFICATIONS

```
generic
     type Key   is limited private;
     type Item  is limited private;
     type Index is (<>);
     type Items is array(Index range <>) of Item;
     with function Is_Equal      (Left  : in Key;
                                  Right : in Item) return
Boolean;
     with function Is_Less_Than (Left  : in Key;
                                  Right : in Item) return
Boolean;
package Ordered_Sequential_Search is

-- modified by Tuan Nguyen
-- 20 Jan 95
-- adding procedures to replace functions

     procedure Location_Of (The_Key      : in Key;
                            In_The_Items : in Items;
                            Result       : out Index);

-- end of modification

     function Location_Of (The_Key      : in Key;
                           In_The_Items : in Items)
return Index;

     Item_Not_Found : exception;

end Ordered_Sequential_Search;
```

ORDERED SEQUENTIAL SEARCH

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Ordered_Sequential_Search is

-- modified by Tuan Nguyen
-- 20 Jan 95
-- adding procedures to replace functions

     procedure Location_Of (The_Key      : in Key;
                            In_The_Items : in Items;
                            Result       : out Index) is
     begin
        Result := Location_Of(The_Key,In_The_Items);
     end Location_Of;

-- end of modification

     function Location_Of (The_Key      : in Key;
                           In_The_Items : in Items)
return Index is
     begin
        for The_Index in In_The_Items'Range loop
            if Is_Equal(The_Key,
In_The_Items(The_Index)) then
                return The_Index;
            elsif Is_Less_Than(The_Key,
In_The_Items(The_Index)) then
                raise Item_Not_Found;
            end if;
        end loop;
        raise Item_Not_Found;
     end Location_Of;

end Ordered_Sequential_Search;
```

ORDERED SEQUENTIAL SEARCH

PSDL

```
OPERATOR Location_Of
SPECIFICATION
  GENERIC
    Key : PRIVATE_TYPE,
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    Is_Equal : FUNCTION[Left : Key, Right : Item,
RETURN : Boolean],
    Is_Less_Than : FUNCTION[Left : Key, Right : Item,
RETURN : Boolean]
  INPUT
    The_Key : Key,
    In_The_Items : Items
  OUTPUT
    Result : Index
  EXCEPTIONS
    Item_Not_Found
END

IMPLEMENTATION ADA Location_Of
END
```

POLYPHASE SORT

ADA SPECIFICATIONS

```
generic
    Number_Of_Files : in Positive;
    type Item is private;
    type File is limited private;
    with procedure Open_For_Reading (The_File : in out
File);
    with procedure Open_For_Writing (The_File : in out
File);
    with procedure Get              (The_File : in out
File;
                                     The_Item : out
Item);
    with procedure Put              (The_File : in out
File;
                                     The_Item : in
Item);
    with procedure Close            (The_File : in out
File);
    with function Next_Item      (From_The_File : in
File) return Item;
    with function "<"            (Left          : in
Item;
                                  Right         : in
Item) return Boolean;
    with function Is_End_Of_File (The_File      : in
File) return Boolean;
package Polyphase_Sort is

    type Files is array (1 .. Number_Of_Files) of File;

    procedure Sort (The_File        : in out File;
                    Temporary_Files : in out Files;
                    Sorted_File     : out    Positive);

    File_Is_Empty : exception;

end Polyphase_Sort;
```

POLYPHASE SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Polyphase_Sort is

    procedure Sort (The_File        : in out File;
                    Temporary_Files : in out Files;
                    Sorted_File     : out    Positive)
is

        Number_Of_Runs             : array (1 ..
Number_Of_Files) of Natural;
        Number_Of_Dummy_Runs       : array (1 ..
Number_Of_Files) of Natural;
        Last_Item                  : array (1 ..
Number_Of_Files) of Item;
        File_Map                   : array (1 ..
Number_Of_Files) of Positive;
        Available_Files            : array (1 ..
Number_Of_Files) of Positive;
        Level                      : Natural := 1;
        Output_File                : Natural := 1;
        Number_Of_Available_Files  : Natural;
        Last_File                  : Positive;
        Last_Runs                  : Natural;
        Last_Dummy_Runs            : Natural;

        procedure Select_File is
            Temporary_Run : Natural;
        begin
            if Number_Of_Dummy_Runs(Output_File) <
               Number_Of_Dummy_Runs(Output_File + 1)
then
                Output_File := Output_File + 1;
            else
                if Number_Of_Dummy_Runs(Output_File) =
0 then
                    Level := Level + 1;
                    Temporary_Run := Number_Of_Runs(1);
                    for Index in 1 .. (Number_Of_Files
- 1) loop
                        Number_Of_Dummy_Runs(Index) :=
                          Temporary_Run +
Number_Of_Runs(Index + 1) -
                          Number_Of_Runs(Index);
                        Number_Of_Runs(Index) :=
                          Temporary_Run +
Number_Of_Runs(Index + 1);
                    end loop;
                end if;
                Output_File := 1;
            end if;
            Number_Of_Dummy_Runs(Output_File) :=
              Number_Of_Dummy_Runs(Output_File) - 1;
        end Select_File;

        procedure Copy_Run is
            Temporary_Item : Item;
        begin
            loop
                Get(The_File, Temporary_Item);
                Put(Temporary_Files(Output_File),
Temporary_Item);
                exit when (Is_End_Of_File(The_File) or
else
                           (Next_Item(The_File) <
Temporary_Item));
            end loop;
            Last_Item(Output_File) := Temporary_Item;
        end Copy_Run;

        procedure Merge_Run is
            File_Index     : Positive;
            Smallest_Item  : Item;
            Smallest_File  : Positive;
            Temporary_Item : Item;
            End_Of_File    : Boolean;
        begin
            loop
                Number_Of_Available_Files := 0;
                for Index in 1 .. (Number_Of_Files - 1)
loop
                    if Number_Of_Dummy_Runs(Index) > 0
then
                        Number_Of_Dummy_Runs(Index) :=
                          Number_Of_Dummy_Runs(Index) -
1;
                    else
                        Number_Of_Available_Files :=
                          Number_Of_Available_Files +
1;
Available_Files(Number_Of_Available_Files) :=
                          File_Map(Index);
                    end if;
                end loop;
                if Number_Of_Available_Files = 0 then
Number_Of_Dummy_Runs(Number_Of_Files) :=
Number_Of_Dummy_Runs(Number_Of_Files) + 1;
                else
                    loop
                        File_Index := 1;
                        Smallest_File := 1;
                        Smallest_Item :=
                          Next_Item
(Temporary_Files(Available_Files(1)));
                        while File_Index <
Number_Of_Available_Files loop
                            File_Index := File_Index +
1;
                            Temporary_Item :=
                              Next_Item
(Temporary_Files(Available_Files(File_Index)));
                            if Temporary_Item <
Smallest_Item then
                                Smallest_Item :=
Temporary_Item;
                                Smallest_File :=
File_Index;
                            end if;
                        end loop;
```

```
Get(Temporary_Files(Available_Files(Smallest_File)),
                         Temporary_Item);
                      End_Of_File :=
                         Is_End_Of_File

(Temporary_Files(Available_Files(Smallest_File)));

Put(Temporary_Files(File_Map(Number_Of_Files)),
                         Temporary_Item);
                      if End_Of_File or else
                         (Next_Item

(Temporary_Files(Available_Files(Smallest_File)))
                         < Temporary_Item) then

Available_Files(Smallest_File) :=

Available_Files(Number_Of_Available_Files);
                         Number_Of_Available_Files
:=

Number_Of_Available_Files - 1;
                      end if;
                      exit when
(Number_Of_Available_Files = 0);
                   end loop;
                end if;
                Last_Runs := Last_Runs - 1;
                exit when (Last_Runs = 0);
            end loop;
        end Merge_Run;

    begin
        for Index in 1 .. (Number_Of_Files - 1) loop
            Number_Of_Runs(Index) := 1;
            Number_Of_Dummy_Runs(Index) := 1;
            Open_For_Writing(Temporary_Files(Index));
        end loop;
        Number_Of_Runs(Number_Of_Files) := 0;
        Number_Of_Dummy_Runs(Number_Of_Files) := 0;
        Open_For_Reading(The_File);
        if Is_End_Of_File(The_File) then
            for Index in 1 .. Number_Of_Files loop
                Close(Temporary_Files(Index));
            end loop;
            Close(The_File);
            raise File_Is_Empty;
        else
            loop
                Select_File;
                Copy_Run;
                exit when (Is_End_Of_File(The_File) or
                           (Output_File =
(Number_Of_Files - 1)));
            end loop;
            while not Is_End_Of_File(The_File) loop
                Select_File;
                if not (Next_Item(The_File) <
Last_Item(Output_File)) then
                    Copy_Run;
                    if Is_End_Of_File(The_File) then

Number_Of_Dummy_Runs(Output_File) :=

Number_Of_Dummy_Runs(Output_File) + 1;
                    else
                        Copy_Run;
                    end if;
                else
                    Copy_Run;
                end if;
            end loop;
            Close(The_File);
            for Index in 1 .. (Number_Of_Files - 1)
loop

Open_For_Reading(Temporary_Files(Index));
            end loop;
            for Index in 1 .. Number_Of_Files loop
                File_Map(Index) := Index;
            end loop;
            loop
                Last_Runs :=
Number_Of_Runs(Number_Of_Files - 1);
                Number_Of_Dummy_Runs(Number_Of_Files)
:= 0;

Open_For_Writing(Temporary_Files(File_Map(Number_Of_Fil
es)));
                Merge_Run;

Open_For_Reading(Temporary_Files(File_Map(Number_Of_Fil
es)));
                Last_File := File_Map(Number_Of_Files);
                Last_Dummy_Runs :=
Number_Of_Dummy_Runs(Number_Of_Files);
                Last_Runs :=
Number_Of_Runs(Number_Of_Files - 1);
                for Index in reverse 2 ..
Number_Of_Files loop
                    File_Map(Index) := File_Map(Index -
1);
                    Number_Of_Runs(Index) :=
                      Number_Of_Runs(Index - 1) -
Last_Runs;
                    Number_Of_Dummy_Runs(Index) :=
                      Number_Of_Dummy_Runs(Index - 1);
                end loop;
                File_Map(1) := Last_File;
                Number_Of_Runs(1) := Last_Runs;
                Number_Of_Dummy_Runs(1) :=
Last_Dummy_Runs;
                Level := Level - 1;
                exit when (Level = 0);
            end loop;
            for Index in 1 .. Number_Of_Files loop
                Close(Temporary_Files(Index));
            end loop;
            Sorted_File := File_Map(1);
        end if;
    end Sort;

end Polyphase_Sort;
```

POLYPHASE SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    File : PRIVATE_TYPE,
    Open_For_Reading : PROCEDURE[The_File : in_out[t :
File]],
    Open_For_Writing : PROCEDURE[The_File : in_out[t :
File]],
    Get : PROCEDURE[The_File : in_out[t : File],
The_Item : out[t : Item]],
    Put : PROCEDURE[The_File : in_out[t : File],
The_Item : in[t : Item]],
    Close : PROCEDURE[The_File : in_out[t : File]],
    Next_Item : FUNCTION[From_The_File : File, RETURN :
Item],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean],
    Is_End_Of_File : FUNCTION[The_File : File, RETURN :
Boolean]
  INPUT
    The_File : File,
    Temporary_Files : Files
  OUTPUT
    The_File : File,
    Temporary_Files : Files,
    Sorted_File : Positive
  EXCEPTIONS
    File_Is_Empty
END

IMPLEMENTATION ADA Sort
END
```

QUICK SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Quick_Sort is

    procedure Sort (The_Items : in out Items);

end Quick_Sort;
```

QUICK SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--           "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Quick_Sort is

    procedure Exchange (Left  : in out Item;
                        Right : in out Item) is
        Temporary_Item : Item;
    begin
        Temporary_Item := Left;
        Left := Right;
        Right := Temporary_Item;
    end Exchange;

    procedure Sort (The_Items : in out Items) is
        procedure Sort_Recursive (Left_Index  : in
Index;
                                  Right_Index : in
Index) is
            Pivot_Item    : Item;
            The_Front     : Index;
            The_Back      : Index;
            Middle_Index  : Index;
        begin
            if Left_Index < Right_Index then
                Middle_Index :=
Index'Val((Index'Pos(Left_Index) +

Index'Pos(Right_Index)) / 2);
                if The_Items(Middle_Index) <
The_Items(Left_Index) then
                    Exchange(The_Items(Middle_Index),
The_Items(Left_Index));
                end if;
                if The_Items(Right_Index) <
The_Items(Left_Index) then
                    Exchange(The_Items(Right_Index),
The_Items(Left_Index));
                end if;
                if The_Items(Right_Index) <
The_Items(Middle_Index) then
                    Exchange(The_Items(Right_Index),
The_Items(Middle_Index));
                end if;
                Pivot_Item := The_Items(Middle_Index);
                Exchange(The_Items(Middle_Index),

The_Items(Index'Pred(Right_Index)));
                The_Front := Index'Succ(Left_Index);
                The_Back := Index'Pred(Right_Index);
                if The_Back /= The_Items'First then
                    The_Back := Index'Pred(The_Back);
                end if;
                loop
                    while The_Items(The_Front) <
Pivot_Item loop
                        The_Front :=
Index'Succ(The_Front);
                    end loop;
                    while Pivot_Item <
The_Items(The_Back) loop
                        The_Back :=
Index'Pred(The_Back);
                    end loop;
                    if The_Front <= The_Back then
                        if (The_Front = The_Items'Last)
or else
                           (The_Back = The_Items'First)
then
                            return;
                        else

Exchange(The_Items(The_Front),

The_Items(The_Back));
                            The_Front :=
Index'Succ(The_Front);
                            The_Back :=
Index'Pred(The_Back);
                        end if;
                    end if;
                    exit when (The_Front > The_Back);
                end loop;
                Sort_Recursive(Left_Index, The_Back);
                Sort_Recursive(The_Front, Right_Index);
            end if;
        end Sort_Recursive;
    begin
        Sort_Recursive(The_Items'First,
The_Items'Last);
    end Sort;

end Quick_Sort;
```

QUICK SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

RADIX SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    Number_Of_Key_Bits : in Positive;
    with function Bit_Of (The_Item : in Item;
                          The_Bit  : in Positive)
    return Boolean;
package Radix_Sort is

    procedure Sort (The_Items : in out Items);

end Radix_Sort;
```

RADIX SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Radix_Sort is

    procedure Sort (The_Items : in out Items) is
        procedure Sort_Recursive (Left_Index  : in
Index;
                                  Right_Index : in
Index;
                                  Bit         : in
Positive) is
            Temporary_Left  : Index;
            Temporary_Right : Index;
            Temporary_Item  : Item;
        begin
            if Right_Index > Left_Index then
                Temporary_Left  := Left_Index;
                Temporary_Right := Right_Index;
                loop
                    while (not
Bit_Of(The_Items(Temporary_Left), Bit)) and
                        (Temporary_Left <
Temporary_Right) loop
                        Temporary_Left :=
Index'Succ(Temporary_Left);
                    end loop;
                    while
(Bit_Of(The_Items(Temporary_Right), Bit)) and
                        (Temporary_Left <
Temporary_Right) loop
                        Temporary_Right :=
Index'Pred(Temporary_Right);
                    end loop;
                    Temporary_Item :=
The_Items(Temporary_Left);
                    The_Items(Temporary_Left) :=
The_Items(Temporary_Right);
                    The_Items(Temporary_Right) :=
Temporary_Item;
                    exit when (Temporary_Left =
Temporary_Right);
                end loop;
                if not Bit_Of(The_Items(Right_Index),
Bit) then
                    Temporary_Right :=
Index'Succ(Temporary_Right);
                end if;
                if Bit < Number_Of_Key_Bits then
                    if Temporary_Right >
The_Items'First then
                        Sort_Recursive
                          (Left_Index,
Index'Pred(Temporary_Right), Bit + 1);
                    end if;
                    Sort_Recursive
                      (Temporary_Right, Right_Index,
Bit + 1);
                end if;
            end if;
        end Sort_Recursive;

    begin
        Sort_Recursive(The_Items'First, The_Items'Last,
1);
    end Sort;

end Radix_Sort;
```

QUICK SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    Bit_Of : FUNCTION[The_Item : Item, The_Bit :
Positive, RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

SEQUENTIAL SEARCH

ADA SPECIFICATIONS

```
generic
    type Key   is limited private;
    type Item  is limited private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function Is_Equal (Left  : in Key;
                            Right : in Item) return
Boolean;
package Sequential_Search is

-- modified by Tuan Nguyen
-- 20 Jan 95
-- adding procedures to replace functions

    procedure Location_Of (The_Key      : in Key;
                           In_The_Items : in Items;
                           Result       : out Index);

-- end of modification

    function Location_Of (The_Key      : in Key;
                          In_The_Items : in Items)
return Index;

    Item_Not_Found : exception;

end Sequential_Search;
```

SEQUENTIAL SEARCH

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Sequential_Search is

-- modified by Tuan Nguyen
-- 20 Jan 95
-- adding procedures to replace functions

    procedure Location_Of (The_Key      : in Key;
                           In_The_Items : in Items;
                           Result       : out Index) is
    begin
        Result := Location_Of(The_Key,In_The_Items);
    end Location_Of;

-- end of modification

    function Location_Of (The_Key      : in Key;
                          In_The_Items : in Items)
return Index is
    begin
        for The_Index in In_The_Items'Range loop
            if Is_Equal(The_Key,
In_The_Items(The_Index)) then
                return The_Index;
            end if;
        end loop;
        raise Item_Not_Found;
    end Location_Of;

end Sequential_Search;
```

SEQUENTIAL SEARCH

PSDL

```
OPERATOR Location_Of
SPECIFICATION
  GENERIC
    Key : PRIVATE_TYPE,
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    Is_Equal : FUNCTION[Left : Key, Right : Item,
RETURN : Boolean]
  INPUT
    The_Key : Key,
    In_The_Items : Items
  OUTPUT
    Result : Index
  EXCEPTIONS
    Item_Not_Found
END

IMPLEMENTATION ADA Location_Of
END
```

SHAKER SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Shaker_Sort is

    procedure Sort (The_Items : in out Items);

end Shaker_Sort;
```

SHAKER SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Shaker_Sort is

    procedure Sort (The_Items : in out Items) is
        Temporary_Item  : Item;
        Temporary_Index : Index;
        Left_Index      : Index;
        Right_Index     : Index;
    begin
        Left_Index := Index'Succ(The_Items'First);
        Right_Index := The_Items'Last;
        loop
            for Middle_Index in reverse Left_Index ..
Right_Index loop
                if The_Items(Middle_Index) <
                  The_Items(Index'Pred(Middle_Index))
then
                    Temporary_Item :=
The_Items(Index'Pred(Middle_Index));
                    The_Items(Index'Pred(Middle_Index))
:=
                      The_Items(Middle_Index);
                    The_Items(Middle_Index) :=
Temporary_Item;
                    Temporary_Index := Middle_Index;
                end if;
            end loop;
            Left_Index := Index'Succ(Temporary_Index);
            for Middle_Index in Left_Index ..
Right_Index loop
                if The_Items(Middle_Index) <
                  The_Items(Index'Pred(Middle_Index))
then
                    Temporary_Item :=
The_Items(Index'Pred(Middle_Index));
                    The_Items(Index'Pred(Middle_Index))
:=
                      The_Items(Middle_Index);
                    The_Items(Middle_Index) :=
Temporary_Item;
                    Temporary_Index := Middle_Index;
                end if;
            end loop;
            Right_Index := Index'Pred(Temporary_Index);
            exit when (Left_Index > Right_Index);
        end loop;
    end Sort;

end Shaker_Sort;
```

SHAKER SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

SHELL SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Shell_Sort is

    procedure Sort (The_Items : in out Items);

end Shell_Sort;
```

SHELL SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Shell_Sort is

    procedure Sort (The_Items : in out Items) is
        Temporary_Item : Item;
        Inner_Index    : Index;
        Increment      : Positive := 1;
    begin
        loop
            exit when (((9 * Increment) + 4) >=
                 (Index'Pos(The_Items'Last) -
Index'Pos(The_Items'First) + 1));
            Increment := (3 * Increment) + 1;
        end loop;
        loop
            for Outer_Index in
              Index'Val(Index'Pos(The_Items'First) +
Increment) ..
              The_Items'Last loop
                Temporary_Item :=
The_Items(Outer_Index);
                Inner_Index := Outer_Index;
                while Temporary_Item <

The_Items(Index'Val(Index'Pos(Inner_Index) -
Increment))
                  loop
                    The_Items(Inner_Index) :=

The_Items(Index'Val(Index'Pos(Inner_Index) -
Increment));
                    Inner_Index :=
                      Index'Val(Index'Pos(Inner_Index)
- Increment);
                    exit when (Index'Pos(Inner_Index) -
Increment <

Index'Pos(The_Items'First));
                end loop;
                The_Items(Inner_Index) :=
Temporary_Item;
            end loop;
            exit when (Increment = 1);
            Increment := (Increment - 1) / 3;
        end loop;
    end Sort;

end Shell_Sort;
```

SHELL SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

STRAIGHT INSERTION SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Straight_Insertion_Sort is

    procedure Sort (The_Items : in out Items);

end Straight_Insertion_Sort;
```

STRAIGHT INSERTION SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--              "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Straight_Insertion_Sort is

    procedure Sort (The_Items : in out Items) is
        Temporary_Item : Item;
        Inner_Index    : Index;
    begin
        for Outer_Index in Index'Succ(The_Items'First)
.. The_Items'Last loop
            Temporary_Item := The_Items(Outer_Index);
            Inner_Index := Outer_Index;
            while Temporary_Item <
The_Items(Index'Pred(Inner_Index)) loop
                The_Items(Inner_Index) :=
The_Items(Index'Pred(Inner_Index));
                Inner_Index := Index'Pred(Inner_Index);
                exit when (Inner_Index =
The_Items'First);
            end loop;
            The_Items(Inner_Index) := Temporary_Item;
        end loop;
    end Sort;

end Straight_Insertion_Sort;
```

STRAIGHT INSERTION SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

STRAIGHT SELECTION SORT

ADA SPECIFICATIONS

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
```

```
package Straight_Selection_Sort is

    procedure Sort (The_Items : in out Items);

end Straight_Selection_Sort;
```

STRAIGHT SELECTION SORT

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--              "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Straight_Selection_Sort is

    procedure Sort (The_Items : in out Items) is
        Temporary_Item  : Item;
        Temporary_Index : Index;
    begin
        for Outer_Index in The_Items'First ..
Index'Pred(The_Items'Last) loop
            Temporary_Index := Outer_Index;
            Temporary_Item := The_Items(Outer_Index);
            for Inner_Index in Index'Succ(Outer_Index)
.. The_Items'Last loop
                if The_Items(Inner_Index) <
Temporary_Item then
                    Temporary_Index := Inner_Index;
                    Temporary_Item :=
The_Items(Inner_Index);
                end if;
            end loop;
            The_Items(Temporary_Index) :=
The_Items(Outer_Index);
            The_Items(Outer_Index) := Temporary_Item;
        end loop;
    end Sort;

end Straight_Selection_Sort;
```

STRAIGHT SELECTION SORT

PSDL

```
OPERATOR Sort
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Index : DISCRETE_TYPE,
    Items : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX :
Index],
    func_"<" : FUNCTION[Left : Item, Right : Item,
RETURN : Boolean]
  INPUT
    The_Items : Items
  OUTPUT
    The_Items : Items
END

IMPLEMENTATION ADA Sort
END
```

STACK OJB3 SPECIFICATION

```
obj STACK[X :: TRIV] is sort Stack .
   protecting NAT .

*** constructors

   op create      : -> Stack .
   op copy        : Stack Stack -> Stack .
   op clear       : Stack -> Stack .
   op push        : Elt Stack -> Stack .
   op pop         : Stack -> Stack .

*** accessors

   op isequal    : Stack Stack -> Bool .
   op depthof    : Stack -> Nat .
   op isempty    : Stack -> Bool .
   op topof      : Stack -> Elt .

*** exceptions

   op overflow    : -> Stack .
   op underflow   : -> Stack .
   op underflow   : -> Elt .

*** variables declaration

   var S S1 : Stack .
   var E E1 : Elt .

*** axioms

   eq clear(S) = create .
   eq copy(S,S1) = S .
   eq pop(create) = underflow .
   eq pop(push(E,S)) = S .
   eq isequal(S,S1) = S == S1 .
   eq depthof(S) = if S == create then 0 else 1 + depthof(pop(S)) fi .
   eq isempty(S) = if S == create then true else false fi .
   eq topof(create) = underflow .
   eq topof(push(E,S)) = E .

endo
```

STACK PROFILE CODES

OPERATORS	*SIGNATURES*	*PROFILE CODES*
COPY	A B -> B	3211
CLEAR	A -> A	2201
PUSH	A B -> B	3211
POP	A -> A	2201
IS_EQUAL	A B -> C	330
DEPTH_OF	A -> B	220
IS_EMPTY	A -> B	220
TOP_OF	A -> B	220

SET OF PROFILE: *{3211,2201,330,220}*

STACK SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA SPECIFICATION

```
generic
    type Item is private;
package Stack_Sequential_Bounded_Managed_Iterator is

    type Stack(The_Size : Positive) is limited private;

    procedure Copy  (From_The_Stack : in     Stack;
                     To_The_Stack   : in out Stack);
    procedure Clear (The_Stack      : in out Stack);
    procedure Push  (The_Item       : in     Item;
                     On_The_Stack   : in out Stack);
    procedure Pop   (The_Stack      : in out Stack);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left      : in Stack;
                        Right     : in Stack;
                        Result    : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal  (Left      : in Stack;
                        Right     : in Stack) return
Boolean;

    function Depth_Of  (The_Stack : in Stack) return
Natural;

    function Is_Empty (The_Stack : in Stack) return
Boolean;

    function Top_Of    (The_Stack : in Stack) return
Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out
Boolean);
    procedure Iterate (Over_The_Stack : in Stack);

    Overflow   : exception;
    Underflow  : exception;

private
    type Items is array(Positive range <>) of Item;
    type Stack(The_Size : Positive) is
        record
            The_Top   : Natural := 0;
            The_Items : Items(1 .. The_Size);
        end record;
end Stack_Sequential_Bounded_Managed_Iterator;
```

STACK SEQUENTIAL BOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--             "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Stack_Sequential_Bounded_Managed_Iterator
is

    procedure Copy (From_The_Stack : in     Stack;
                    To_The_Stack   : in out Stack) is
    begin
        if From_The_Stack.The_Top >
To_The_Stack.The_Size then
            raise Overflow;
        else
            To_The_Stack.The_Items(1 ..
From_The_Stack.The_Top) :=
                From_The_Stack.The_Items(1 ..
From_The_Stack.The_Top);
            To_The_Stack.The_Top :=
From_The_Stack.The_Top;
        end if;
    end Copy;

    procedure Clear (The_Stack : in out Stack) is
    begin
        The_Stack.The_Top := 0;
    end Clear;

    procedure Push (The_Item     : in     Item;
                    On_The_Stack : in out Stack) is
    begin
        On_The_Stack.The_Items(On_The_Stack.The_Top +
1) := The_Item;
        On_The_Stack.The_Top := On_The_Stack.The_Top +
1;
    exception
        when Constraint_Error =>
            raise Overflow;
    end Push;

    procedure Pop (The_Stack : in out Stack) is
    begin
        The_Stack.The_Top := The_Stack.The_Top - 1;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing procedures with functions

    procedure Is_Equal (Left   : in Stack;
                        Right  : in Stack;
                        Result : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Depth_Of (The_Stack : in Stack;
                        Result    : out Natural) is
    begin
        Result := Depth_Of(The_Stack);
    end Depth_Of;

    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean) is
    begin
        Result := Is_Empty(The_Stack);
    end Is_Empty;

    procedure Top_Of (The_Stack : in Stack;
                      Result    : out Item) is
    begin
        Result := Top_Of(The_Stack);
    end Top_Of;

--  end of modification

    function Is_Equal (Left  : in Stack;
                       Right : in Stack) return Boolean
is
    begin
        if Left.The_Top /= Right.The_Top then
            return False;
        else
            for Index in 1 .. Left.The_Top loop
                if Left.The_Items(Index) /=
Right.The_Items(Index) then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Depth_Of (The_Stack : in Stack) return
Natural is
    begin
        return The_Stack.The_Top;
    end Depth_Of;

    function Is_Empty (The_Stack : in Stack) return
Boolean is
    begin
        return (The_Stack.The_Top = 0);
    end Is_Empty;

    function Top_Of (The_Stack : in Stack) return Item
is
    begin
        return The_Stack.The_Items(The_Stack.The_Top);
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    procedure Iterate (Over_The_Stack : in Stack) is
        Continue : Boolean;
    begin
        for The_Iterator in reverse 1 ..
Over_The_Stack.The_Top loop

Process(Over_The_Stack.The_Items(The_Iterator),
Continue);
            exit when not Continue;
        end loop;
    end Iterate;

end Stack_Sequential_Bounded_Managed_Iterator;
```

STACK SEQUENTIAL BOUNDED MANAGED ITERATOR

PSDL

```
TYPE Stack_Sequential_Bounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Stack,
      Right : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

 OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
    INPUT
      Over_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Stack_Sequential_Bounded_Managed_Iterator
END
```

STACK SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
package Stack_Sequential_Unbounded_Managed_Noniterator
is

    type Stack is limited private;

    procedure Copy  (From_The_Stack : in     Stack;
                     To_The_Stack   : in out Stack);
    procedure Clear (The_Stack      : in out Stack);
    procedure Push  (The_Item       : in     Item;
                     On_The_Stack   : in out Stack);
    procedure Pop   (The_Stack      : in out Stack);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left      : in Stack;
                        Right     : in Stack;
                        Result    : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left      : in Stack;
                       Right     : in Stack) return
Boolean;
    function Depth_Of (The_Stack : in Stack) return
Natural;
    function Is_Empty (The_Stack : in Stack) return
Boolean;
    function Top_Of   (The_Stack : in Stack) return
Item;

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Stack is access Node;
end Stack_Sequential_Unbounded_Managed_Noniterator;
```

STACK SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Stack_Sequential_Unbounded_Managed_Noniterator is

    type Node is
        record
            The_Item : Item;
            Next     : Stack;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Stack) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Stack
is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new
Storage_Manager_Sequential
                                   (Item        =>
Node,
                                    Pointer     =>
Stack,
                     Free        => Free,
                     Set_Pointer => Set_Next,
                     Pointer_Of  => Next_Of);

    procedure Copy (From_The_Stack : in     Stack;
                    To_The_Stack   : in out Stack) is
        From_Index : Stack := From_The_Stack;
        To_Index   : Stack;
    begin
        Node_Manager.Free(To_The_Stack);
        if From_The_Stack /= null then
            To_The_Stack := Node_Manager.New_Item;
            To_The_Stack.The_Item :=
From_Index.The_Item;
            To_Index := To_The_Stack;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index := To_Index.Next;
                To_Index.The_Item :=
From_Index.The_Item;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Stack : in out Stack) is
    begin
        Node_Manager.Free(The_Stack);
    end Clear;

    procedure Push (The_Item     : in     Item;
                    On_The_Stack : in out Stack) is
        Temporary_Node : Stack;
    begin
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.Next := On_The_Stack;
        On_The_Stack := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Push;

    procedure Pop (The_Stack : in out Stack) is
        Temporary_Node : Stack;
    begin
        Temporary_Node := The_Stack;
        The_Stack := Temporary_Node.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left      : in Stack;
                        Right     : in Stack;
                        Result    : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left  : in Stack;
                       Right : in Stack) return Boolean
is
        Left_Index  : Stack := Left;
        Right_Index : Stack := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /=
Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Depth_Of (The_Stack : in Stack) return
Natural is
        Count : Natural := 0;
        Index : Stack   := The_Stack;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Depth_Of;

    function Is_Empty (The_Stack : in Stack) return
Boolean is
    begin
        return (The_Stack = null);
    end Is_Empty;

    function Top_Of (The_Stack : in Stack) return Item
is
    begin
        return The_Stack.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

end Stack_Sequential_Unbounded_Managed_Noniterator;
```

STACK SEQUENTIAL UNBOUNDED MANAGED NONITERATOR

PSDL

```
TYPE Stack_Sequential_Unbounded_Managed_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Stack,
      Right : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Stack_Sequential_Unbounded_Managed_Noniterator
END
```

STACK SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATION

```
generic
    type Item is private;
package Stack_Sequential_Unbounded_Managed_Iterator is

    type Stack is limited private;

    procedure Copy  (From_The_Stack : in     Stack;
                     To_The_Stack   : in out Stack);
    procedure Clear (The_Stack      : in out Stack);
    procedure Push  (The_Item       : in     Item;
                     On_The_Stack   : in out Stack);
    procedure Pop   (The_Stack      : in out Stack);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left       : in Stack;
                        Right      : in Stack;
                        Result     : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result     : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left       : in Stack;
                       Right      : in Stack) return
Boolean;
    function Depth_Of (The_Stack : in Stack) return
Natural;
    function Is_Empty (The_Stack : in Stack) return
Boolean;
    function Top_Of   (The_Stack : in Stack) return
Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out
Boolean);
    procedure Iterate (Over_The_Stack : in Stack);

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Stack is access Node;
end Stack_Sequential_Unbounded_Managed_Iterator;
```

STACK SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body
Stack_Sequential_Unbounded_Managed_Iterator is

    type Node is
        record
            The_Item : Item;
            Next     : Stack;
        end record;

    procedure Free (The_Node : in out Node) is
    begin
        null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Stack) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Stack
is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new
Storage_Manager_Sequential
                                        (Item        =>
Node,
                                         Pointer     =>
Stack,
                         Free       => Free,
                         Set_Pointer => Set_Next,
                         Pointer_Of  => Next_Of);

    procedure Copy (From_The_Stack : in     Stack;
                    To_The_Stack   : in out Stack) is
        From_Index : Stack := From_The_Stack;
        To_Index   : Stack;
    begin
        Node_Manager.Free(To_The_Stack);
        if From_The_Stack /= null then
            To_The_Stack := Node_Manager.New_Item;
            To_The_Stack.The_Item :=
From_Index.The_Item;
            To_Index := To_The_Stack;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := Node_Manager.New_Item;
                To_Index := To_Index.Next;
                To_Index.The_Item :=
From_Index.The_Item;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Stack : in out Stack) is
    begin
        Node_Manager.Free(The_Stack);
    end Clear;

    procedure Push (The_Item     : in     Item;
                    On_The_Stack : in out Stack) is
        Temporary_Node : Stack;
    begin
        Temporary_Node := Node_Manager.New_Item;
        Temporary_Node.The_Item := The_Item;
        Temporary_Node.Next := On_The_Stack;
        On_The_Stack := Temporary_Node;
    exception
        when Storage_Error =>
            raise Overflow;
    end Push;

    procedure Pop (The_Stack : in out Stack) is
        Temporary_Node : Stack;
    begin
        Temporary_Node := The_Stack;
        The_Stack := Temporary_Node.Next;
        Temporary_Node.Next := null;
        Node_Manager.Free(Temporary_Node);
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left      : in Stack;
                        Right     : in Stack;
                        Result    : out Boolean);
    procedure Depth_Of (The_Stack : in Stack;
                        Result    : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left  : in Stack;
                       Right : in Stack) return Boolean
is
        Left_Index  : Stack := Left;
        Right_Index : Stack := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /=
Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Depth_Of (The_Stack : in Stack) return
Natural is
        Count : Natural := 0;
        Index : Stack   := The_Stack;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Depth_Of;

    function Is_Empty (The_Stack : in Stack) return
Boolean is
    begin
        return (The_Stack = null);
    end Is_Empty;

    function Top_Of (The_Stack : in Stack) return Item
is
    begin
        return The_Stack.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    procedure Iterate (Over_The_Stack : in Stack) is
        The_Iterator : Stack := Over_The_Stack;
        Continue     : Boolean;
    begin
        while not (The_Iterator = null) loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Stack_Sequential_Unbounded_Managed_Iterator;
```

STACK SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE Stack_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Stack,
      Right : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
    INPUT
      Over_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Stack_Sequential_Unbounded_Managed_Iterator
END
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATION

```
generic
    type Item is private;
package
Stack_Sequential_Unbounded_Unmanaged_Noniterator is

    type Stack is limited private;

    procedure Copy  (From_The_Stack : in     Stack;
                     To_The_Stack   : in out Stack);
    procedure Clear (The_Stack      : in out Stack);
    procedure Push  (The_Item       : in     Item;
                     On_The_Stack   : in out Stack);
    procedure Pop   (The_Stack      : in out Stack);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left      : in Stack;
                        Right     : in Stack;
                        Result    : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left      : in Stack;
                       Right     : in Stack) return
Boolean;
    function Depth_Of (The_Stack : in Stack) return
Natural;
    function Is_Empty (The_Stack : in Stack) return
Boolean;
    function Top_Of   (The_Stack : in Stack) return
Item;

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Stack is access Node;
end Stack_Sequential_Unbounded_Unmanaged_Noniterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION


```
package body
Stack_Sequential_Unbounded_Unmanaged_Noniterator is

    type Node is
        record
            The_Item : Item;
            Next     : Stack;
        end record;

    procedure Copy (From_The_Stack : in     Stack;
                    To_The_Stack   : in out Stack) is
        From_Index : Stack := From_The_Stack;
        To_Index   : Stack;
    begin
        if From_The_Stack = null then
            To_The_Stack := null;
        else
            To_The_Stack := new Node'(The_Item =>
From_Index.The_Item,
                                      Next     =>
null);
            To_Index := To_The_Stack;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     =>
null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Stack : in out Stack) is
    begin
        The_Stack := null;
    end Clear;

    procedure Push (The_Item     : in     Item;
                    On_The_Stack : in out Stack) is
    begin
        On_The_Stack := new Node'(The_Item => The_Item,
                                  Next     =>
On_The_Stack);
    exception
        when Storage_Error =>
            raise Overflow;
    end Push;

    procedure Pop (The_Stack : in out Stack) is
    begin
        The_Stack := The_Stack.Next;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left      : in Stack;
                        Right     : in Stack;
                        Result    : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left  : in Stack;
                       Right : in Stack) return Boolean
is
        Left_Index  : Stack := Left;
        Right_Index : Stack := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /=
Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Depth_Of (The_Stack : in Stack) return
Natural is
        Count : Natural := 0;
        Index : Stack   := The_Stack;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Depth_Of;

    function Is_Empty (The_Stack : in Stack) return
Boolean is
    begin
        return (The_Stack = null);
    end Is_Empty;

    function Top_Of (The_Stack : in Stack) return Item
is
    begin
        return The_Stack.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

end Stack_Sequential_Unbounded_Unmanaged_Noniterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE Stack_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Stack,
      Right : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Stack_Sequential_Unbounded_Unmanaged_Noniterator
END
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATION

```
generic
    type Item is private;
package Stack_Sequential_Unbounded_Unmanaged_Iterator
is

    type Stack is limited private;

    procedure Copy  (From_The_Stack : in     Stack;
                     To_The_Stack   : in out Stack);
    procedure Clear (The_Stack      : in out Stack);
    procedure Push  (The_Item       : in     Item;
                     On_The_Stack   : in out Stack);
    procedure Pop   (The_Stack      : in out Stack);

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left       : in Stack;
                        Right      : in Stack;
                        Result     : out Boolean);
    procedure Depth_Of (The_Stack   : in Stack;
                        Result      : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result     : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left      : in Stack;
                       Right     : in Stack) return
Boolean;
    function Depth_Of (The_Stack : in Stack) return
Natural;
    function Is_Empty (The_Stack : in Stack) return
Boolean;
    function Top_Of   (The_Stack : in Stack) return
Item;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out
Boolean);
    procedure Iterate (Over_The_Stack : in Stack);

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Stack is access Node;
end Stack_Sequential_Unbounded_Unmanaged_Iterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body
Stack_Sequential_Unbounded_Unmanaged_Iterator is

    type Node is
        record
            The_Item : Item;
            Next     : Stack;
        end record;

    procedure Copy (From_The_Stack : in     Stack;
                    To_The_Stack   : in out Stack) is
        From_Index : Stack := From_The_Stack;
        To_Index   : Stack;
    begin
        if From_The_Stack = null then
            To_The_Stack := null;
        else
            To_The_Stack := new Node'(The_Item =>
From_Index.The_Item,
                                      Next     =>
null);
            To_Index := To_The_Stack;
            From_Index := From_Index.Next;
            while From_Index /= null loop
                To_Index.Next := new Node'(The_Item =>
From_Index.The_Item,
                                           Next     =>
null);
                To_Index := To_Index.Next;
                From_Index := From_Index.Next;
            end loop;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Stack : in out Stack) is
    begin
        The_Stack := null;
    end Clear;

    procedure Push (The_Item     : in     Item;
                    On_The_Stack : in out Stack) is
    begin
        On_The_Stack := new Node'(The_Item => The_Item,
                                  Next     =>
On_The_Stack);
    exception
        when Storage_Error =>
            raise Overflow;
    end Push;

    procedure Pop (The_Stack : in out Stack) is
    begin
        The_Stack := The_Stack.Next;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Pop;

--  modified by Tuan Nguyen
--  replacing functions with procedures

    procedure Is_Equal (Left      : in Stack;
                        Right     : in Stack;
                        Result    : out Boolean);
    procedure Depth_Of (The_Stack  : in Stack;
                        Result     : out Natural);
    procedure Is_Empty (The_Stack : in Stack;
                        Result    : out Boolean);
    procedure Top_Of (The_Stack  : in Stack;
                      Result     : out Item);

--  end of modification

    function Is_Equal (Left  : in Stack;
                       Right : in Stack) return Boolean
is
        Left_Index  : Stack := Left;
        Right_Index : Stack := Right;
    begin
        while Left_Index /= null loop
            if Left_Index.The_Item /=
Right_Index.The_Item then
                return False;
            end if;
            Left_Index := Left_Index.Next;
            Right_Index := Right_Index.Next;
        end loop;
        return (Right_Index = null);
    exception
        when Constraint_Error =>
            return False;
    end Is_Equal;

    function Depth_Of (The_Stack : in Stack) return
Natural is
        Count : Natural := 0;
        Index : Stack   := The_Stack;
    begin
        while Index /= null loop
            Count := Count + 1;
            Index := Index.Next;
        end loop;
        return Count;
    end Depth_Of;

    function Is_Empty (The_Stack : in Stack) return
Boolean is
    begin
        return (The_Stack = null);
    end Is_Empty;

    function Top_Of (The_Stack : in Stack) return Item
is
    begin
        return The_Stack.The_Item;
    exception
        when Constraint_Error =>
            raise Underflow;
    end Top_Of;

    procedure Iterate (Over_The_Stack : in Stack) is
        The_Iterator : Stack := Over_The_Stack;
        Continue     : Boolean;
    begin
        while not (The_Iterator = null) loop
            Process(The_Iterator.The_Item, Continue);
            exit when not Continue;
            The_Iterator := The_Iterator.Next;
        end loop;
    end Iterate;

end Stack_Sequential_Unbounded_Unmanaged_Iterator;
```

STACK SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE Stack_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Stack : Stack,
      To_The_Stack : Stack
    OUTPUT
      To_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Push
  SPECIFICATION
    INPUT
      The_Item : Item,
      On_The_Stack : Stack
    OUTPUT
      On_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Pop
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Stack,
      Right : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Depth_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Is_Empty
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Top_Of
  SPECIFICATION
    INPUT
      The_Stack : Stack
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Underflow
  END

  OPERATOR Iterate
  SPECIFICATION
    GENERIC
      Process : PROCEDURE[The_Item : in[t : Item],
Continue : out[t : Boolean]]
    INPUT
      Over_The_Stack : Stack
    EXCEPTIONS
      Overflow, Underflow
  END

END
IMPLEMENTATION ADA
Stack_Sequential_Unbounded_Unmanaged_Iterator
END
```

STORAGE MANAGER SEQUENTIAL

ADA SPECIFICATION

```
generic
    type Item is limited private;
    type Pointer is access Item;
    with procedure Free        (The_Item    : in out
Item);
    with procedure Set_Pointer (The_Item    : in out
Item;
                                The_Pointer : in
Pointer);
    with function Pointer_Of (The_Item : in Item)
return Pointer;
package Storage_Manager_Sequential is

    procedure Free (The_Pointer : in out Pointer);

--  modified by Tuan Nguyen
--  replace function with procedure

    procedure New_Item (Result : Pointer);

--  end of modification

    function New_Item return Pointer;

end Storage_Manager_Sequential;
```

STORAGE MANAGER SEQUENTIAL

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady
Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3)
(ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body Storage_Manager_Sequential is

    Free_List : Pointer := null;

    procedure Free (The_Pointer : in out Pointer) is
        Temporary_Pointer : Pointer;
    begin
        while The_Pointer /= null loop
            Temporary_Pointer := The_Pointer;
            The_Pointer := Pointer_Of(The_Pointer.all);
            Free(Temporary_Pointer.all);
            Set_Pointer(Temporary_Pointer.all,
The_Pointer => Free_List);
            Free_List := Temporary_Pointer;
        end loop;
    end Free;

--  modified by Tuan Nguyen
--  replace function with procedure

    procedure New_Item (Result : Pointer) is
    begin
        Result := New_Item;
    end New_Item;

--  end of modification

    function New_Item return Pointer is
        Temporary_Pointer : Pointer;
    begin
        if Free_List = null then
            return new Item;
        else
            Temporary_Pointer := Free_List;
            Free_List :=
Pointer_Of(Temporary_Pointer.all);
            Set_Pointer(Temporary_Pointer.all,
The_Pointer => null);
            return Temporary_Pointer;
        end if;
    end New_Item;

end Storage_Manager_Sequential;
```

STORAGE MANAGER SEQUENTIAL

PSDL

```
OPERATOR Free
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Pointer : ACCESS_TYPE,
    Free : PROCEDURE[The_Item : in_out[t : Item]],
    Set_Pointer : PROCEDURE[The_Item : in_out[t :
Item], The_Pointer : in[t : Pointer]],
    Pointer_Of : FUNCTION[The_Item : Item, RETURN :
Pointer]
  INPUT
    The_Pointer : Pointer
  OUTPUT
    The_Pointer : Pointer
END

IMPLEMENTATION ADA Free
END
```

STRING SEQUENTIAL UNBOUNDED CONTROLLED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Substring is array(Positive range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
package String_Sequential_Unbounded_Controlled_Iterator is

    type String is limited private;

    procedure Copy      (From_The_String    : in     String;
                         To_The_String      : in out String);
    procedure Copy      (From_The_Substring : in     Substring;
                         To_The_String      : in out String);
    procedure Clear     (The_String         : in out String);
    procedure Prepend   (The_String         : in     String;
                         To_The_String      : in out String);
    procedure Prepend   (The_Substring      : in     Substring;
                         To_The_String      : in out String);
    procedure Append    (The_String         : in     String;
                         To_The_String      : in out String);
    procedure Append    (The_Substring      : in     Substring;
                         To_The_String      : in out String);
    procedure Insert    (The_String         : in     String;
                         In_The_String      : in out String;
                         At_The_Position    : in     Positive);
    procedure Insert    (The_Substring      : in     Substring;
                         In_The_String      : in out String;
                         At_The_Position    : in     Positive);
    procedure Delete    (In_The_String      : in out String;
                         From_The_Position  : in     Positive;
                         To_The_Position    : in     Positive);
    procedure Replace   (In_The_String      : in out String;
                         At_The_Position    : in     Positive;
                         With_The_String    : in     String);
    procedure Replace   (In_The_String      : in out String;
                         At_The_Position    : in     Positive;
                         With_The_Substring : in     Substring);
    procedure Set_Item  (In_The_String      : in out String;
                         At_The_Position    : in     Positive;
                         With_The_Item      : in     Item);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal          (Left              : in String;
                                 Right             : in String;
                                 Result            : out Boolean);
    procedure Is_Equal          (Left              : in Substring;
                                 Right             : in String;
                                 Result            : out Boolean);
    procedure Is_Equal          (Left              : in String;
                                 Right             : in Substring;
                                 Result            : out Boolean);
    procedure Is_Less_Than      (Left              : in String;
                                 Right             : in String;
                                 Result            : out Boolean);
    procedure Is_Less_Than      (Left              : in Substring;
                                 Right             : in String;
                                 Result            : out Boolean);
    procedure Is_Less_Than      (Left              : in String;
                                 Right             : in Substring;
                                 Result            : out Boolean);
    procedure Is_Greater_Than   (Left              : in String;
                                 Right             : in String;
                                 Result            : out Boolean);
    procedure Is_Greater_Than   (Left              : in Substring;
                                 Right             : in String;
                                 Result            : out Boolean);
    procedure Is_Greater_Than   (Left              : in String;
                                 Right             : in Substring;
                                 Result            : out Boolean);
    procedure Length_Of         (The_String        : in String;
                                 Result            : out Natural);
    procedure Is_Null           (The_String        : in String;
                                 Result            : out Boolean);
    procedure Item_Of           (The_String        : in String;
                                 At_The_Position   : in Positive;
                                 Result            : out Item);
    procedure Substring_Of      (The_String        : in String;
                                 Result            : out Substring);
    procedure Substring_Of      (The_String        : in String;
                                 From_The_Position : in Positive;
                                 To_The_Position   : in Positive;
                                 Result            : out Substring);

--  end of modification

    function Is_Equal        (Left            : in String;
                              Right           : in String)     return
Boolean;
    function Is_Equal        (Left            : in Substring;
                              Right           : in String)     return
Boolean;
    function Is_Equal        (Left            : in String;
                              Right           : in Substring)  return
Boolean;
    function Is_Less_Than    (Left            : in String;
                              Right           : in String)     return
Boolean;
    function Is_Less_Than    (Left            : in Substring;
                              Right           : in String)     return
Boolean;
    function Is_Less_Than    (Left            : in String;
                              Right           : in Substring)  return
Boolean;
    function Is_Greater_Than (Left            : in String;
                              Right           : in String)     return
Boolean;
    function Is_Greater_Than (Left            : in Substring;
                              Right           : in String)     return
Boolean;
    function Is_Greater_Than (Left            : in String;
                              Right           : in Substring)  return
Boolean;
    function Length_Of       (The_String      : in String)     return
Natural;
    function Is_Null         (The_String      : in String)     return
Boolean;
    function Item_Of         (The_String      : in String;
                              At_The_Position : in Positive) return
Item;
    function Substring_Of    (The_String      : in String)   return
Substring;
    function Substring_Of    (The_String        : in String;
                              From_The_Position : in Positive;
                              To_The_Position   : in Positive)return
Substring;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_String : in String);

    Overflow       : exception;
    Position_Error : exception;

private
    type Structure is access Substring;
    type String is
        record
            The_Length : Natural := 0;
            The_Items  : Structure;
        end record;
end String_Sequential_Unbounded_Controlled_Iterator;
```

STRING SEQUENTIAL UNBOUNDED CONTROLLED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Storage_Manager_Sequential;
package body String_Sequential_Unbounded_Controlled_Iterator is

    type Node;
    type Node_Pointer is access Node;
    type Node is
        record
            The_Structure : Structure;
            Next          : Node_Pointer;
        end record;

    type Header;
    type Header_Pointer is access Header;
    type Header is
        record
            The_Size       : Natural;
            The_Structures : Node_Pointer;
            Next           : Header_Pointer;
        end record;

    procedure Free(The_Node : in out Node) is
    begin
        The_Node.The_Structure := null;
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Node_Pointer) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Node_Pointer is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Node_Pointer,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Free(The_Header : in out Header) is
    begin
        The_Header.The_Size := 0;
    end Free;

    procedure Set_Next (The_Header : in out Header;
                        To_Next    : in     Header_Pointer) is
    begin
        The_Header.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Header : in Header) return Header_Pointer is
    begin
        return The_Header.Next;
    end Next_Of;

    package Header_Manager is new Storage_Manager_Sequential
                                  (Item        => Header,
                                   Pointer     => Header_Pointer,
                                   Free        => Free,
                                   Set_Pointer => Set_Next,
                                   Pointer_Of  => Next_Of);

    task Structure_Manager is
        entry Free              (The_Structure : in out Structure);
        entry Get_New_Structure (The_Size      : in Natural;
                                 The_Structure : out Structure);
    end Structure_Manager;

    task body Structure_Manager is
        Free_List       : Header_Pointer;
        The_Structure   : Structure;
        Node_Index      : Node_Pointer;
        Previous_Header : Header_Pointer;
        Header_Index    : Header_Pointer;
    begin
        loop
            begin
                select
                    accept Free (The_Structure : in out Structure) do
                        Previous_Header := null;
                        Header_Index := Free_List;
                        while Header_Index /= null loop
                            if The_Structure'Length <
                              Header_Index.The_Size then
                                exit;
                            elsif The_Structure'Length =
                              Header_Index.The_Size then
                                Node_Index := Node_Manager.New_Item;
                                Node_Index.The_Structure :=
The_Structure;
                                Node_Index.Next :=
Header_Index.The_Structures;
                                Header_Index.The_Structures :=
Node_Index;
                                The_Structure := null;
                                return;
                            end if;
                            Previous_Header := Header_Index;
                            Header_Index := Header_Index.Next;
                        end loop;
                        Header_Index := Header_Manager.New_Item;
                        Header_Index.The_Size := The_Structure'Length;
                        Node_Index := Node_Manager.New_Item;
                        Node_Index.The_Structure := The_Structure;
                        Header_Index.The_Structures := Node_Index;
                        if Previous_Header = null then
                            Header_Index.Next := Free_List;
                            Free_List := Header_Index;
                        else
                            Header_Index.Next := Previous_Header.Next;
                            Previous_Header.Next := Header_Index;
                        end if;
                        The_Structure := null;
                    end Free;
                or
                    accept Get_New_Structure (The_Size      : in
Natural;
                                              The_Structure : out
Structure) do
                        Previous_Header := null;
                        Header_Index := Free_List;
                        while Header_Index /= null loop
                            if Header_Index.The_Size >= The_Size then
                                Node_Index :=
Header_Index.The_Structures;
                                Header_Index.The_Structures :=
Node_Index.Next;
                                Node_Index.Next := null;
                                if Header_Index.The_Structures = null
then
                                    if Previous_Header = null then
                                        Free_List :=
Header_Index.Next;
                                    else
                                        Previous_Header.Next :=
                                          Header_Index.Next;
                                    end if;
                                    Header_Index.Next := null;
                                    Header_Manager.Free(Header_Index);
                                end if;
                                The_Structure :=
Node_Index.The_Structure;
                                Node_Manager.Free(Node_Index);
                                return;
                            end if;
                            Previous_Header := Header_Index;
                            Header_Index := Header_Index.Next;
                        end loop;
                        The_Structure := new Substring(1 .. The_Size);
                    end Get_New_Structure;
                or
                    terminate;
                end select;
            exception
                when Storage_Error =>
                    null;
            end;
        end loop;
    end Structure_Manager;

    procedure Free (The_Structure : in out Structure) is
    begin
        if The_Structure /= null then
            Structure_Manager.Free(The_Structure);
        end if;
    end Free;

    function New_Structure (The_Size : in Natural) return Structure is
        Temporary_Structure : Structure;
    begin
        Structure_Manager.Get_New_Structure(The_Size,
Temporary_Structure);
        return Temporary_Structure;
    end New_Structure;

    procedure Set (The_String        : in out String;
                   To_The_Size       : in     Natural;
                   Preserve_The_Value : in    Boolean) is
        Temporary_Structure : Structure;
    begin
```

```
        if To_The_Size = 0 then
            Free(The_String.The_Items);
        elsif The_String.The_Items = null then
            The_String.The_Items := New_Structure(The_Size =>
To_The_Size);
        elsif To_The_Size > The_String.The_Items'Length then
            if Preserve_The_Value then
                Temporary_Structure := New_Structure(To_The_Size);
                Temporary_Structure(1 .. The_String.The_Length) :=
                  The_String.The_Items(1 .. The_String.The_Length);
                Free(The_String.The_Items);
                The_String.The_Items := Temporary_Structure;
            else
                Free(The_String.The_Items);
                The_String.The_Items := New_Structure
                                          (The_Size => To_The_Size);
            end if;
        end if;
        The_String.The_Length := To_The_Size;
    exception
        when Storage_Error =>
            raise Overflow;
    end Set;

    procedure Copy (From_The_String : in     String;
                    To_The_String   : in out String) is
    begin
        Set(To_The_String,
            To_The_Size        => From_The_String.The_Length,
            Preserve_The_Value => False);
        To_The_String.The_Items(1 .. From_The_String.The_Length) :=
          From_The_String.The_Items(1 .. From_The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Copy (From_The_Substring : in     Substring;
                    To_The_String      : in out String) is
    begin
        Set(To_The_String,
            To_The_Size        => From_The_Substring'Length,
            Preserve_The_Value => False);
        To_The_String.The_Items(1 .. From_The_Substring'Length) :=
          From_The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_String : in out String) is
    begin
        Set(The_String,
            To_The_Size        => 0,
            Preserve_The_Value => False);
    exception
        when Storage_Error =>
            raise Overflow;
    end Clear;

    procedure Prepend (The_String    : in     String;
                       To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_String.The_Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((The_String.The_Length + 1) ..
New_Length)
          := To_The_String.The_Items(1 .. Old_Length);
        To_The_String.The_Items(1 .. The_String.The_Length) :=
          The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Prepend;

    procedure Prepend (The_Substring : in     Substring;
                       To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_Substring'Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((The_Substring'Length + 1) ..
New_Length)
          := To_The_String.The_Items(1 .. Old_Length);
        To_The_String.The_Items(1 .. The_Substring'Length) :=
The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Prepend;

    procedure Append (The_String    : in     String;
                      To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_String.The_Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((Old_Length + 1) .. New_Length)
          := The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Append;

    procedure Append (The_Substring : in     Substring;
                      To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_Substring'Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((Old_Length + 1) .. New_Length)
          := The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Append;

    procedure Insert (The_String      : in     String;
                      In_The_String   : in out String;
                      At_The_Position : in     Positive) is
        Old_Length   : Natural := In_The_String.The_Length;
        New_Length   : Natural :=
                         In_The_String.The_Length +
The_String.The_Length;
        End_Position : Natural :=
                         At_The_Position + The_String.The_Length;
    begin
        if At_The_Position > In_The_String.The_Length then
            raise Position_Error;
        else
            Set(In_The_String,
                To_The_Size        => New_Length,
                Preserve_The_Value => True);
            In_The_String.The_Items(End_Position .. New_Length) :=
              In_The_String.The_Items(At_The_Position .. Old_Length);
            In_The_String.The_Items(At_The_Position .. (End_Position -
1)) :=
              The_String.The_Items(1 .. The_String.The_Length);
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Insert (The_Substring   : in     Substring;
                      In_The_String   : in out String;
                      At_The_Position : in     Positive) is
        Old_Length   : Natural := In_The_String.The_Length;
        New_Length   : Natural :=
                         In_The_String.The_Length +
The_Substring'Length;
        End_Position : Natural :=
                         At_The_Position + The_Substring'Length;
    begin
        if At_The_Position > In_The_String.The_Length then
            raise Position_Error;
        else
            Set(In_The_String,
                To_The_Size        => New_Length,
                Preserve_The_Value => True);
            In_The_String.The_Items(End_Position .. New_Length) :=
              In_The_String.The_Items(At_The_Position .. Old_Length);
            In_The_String.The_Items(At_The_Position .. (End_Position -
1)) :=
              The_Substring;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Delete (In_The_String     : in out String;
                      From_The_Position : in     Positive;
                      To_The_Position   : in     Positive) is
        New_Length : Natural;
    begin
        if (From_The_Position > In_The_String.The_Length) or else
           (To_The_Position > In_The_String.The_Length) or else
           (From_The_Position > To_The_Position) then
            raise Position_Error;
        else
            New_Length := In_The_String.The_Length -
                            (To_The_Position - From_The_Position + 1);
            In_The_String.The_Items(From_The_Position .. New_Length)
:=
              In_The_String.The_Items
              ((To_The_Position + 1) .. In_The_String.The_Length);
            Set(In_The_String,
                To_The_Size        => New_Length,
                Preserve_The_Value => True);
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Delete;

    procedure Replace (In_The_String   : in out String;
                       At_The_Position : in     Positive;
                       With_The_String : in     String) is
        End_Position : Natural :=
                         At_The_Position + With_The_String.The_Length -
1;
    begin
```

```
        if (At_The_Position > In_The_String.The_Length) or else
           (End_Position > In_The_String.The_Length) then
            raise Position_Error;
        else
            In_The_String.The_Items(At_The_Position .. End_Position)
:=
              With_The_String.The_Items(1 ..
With_The_String.The_Length);
        end if;
    end Replace;

    procedure Replace (In_The_String      : in out String;
                       At_The_Position    : in     Positive;
                       With_The_Substring : in     Substring) is
        End_Position : Natural :=
                       At_The_Position + With_The_Substring'Length -
1;
    begin
        if (At_The_Position > In_The_String.The_Length) or else
           (End_Position > In_The_String.The_Length) then
            raise Position_Error;
        else
            In_The_String.The_Items(At_The_Position .. End_Position)
:=
              With_The_Substring;
        end if;
    end Replace;

    procedure Set_Item (In_The_String   : in out String;
                        At_The_Position : in     Positive;
                        With_The_Item   : in     Item) is
    begin
        if At_The_Position > In_The_String.The_Length then
            raise Position_Error;
        else
            In_The_String.The_Items(At_The_Position) := With_The_Item;
        end if;
    end Set_Item;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal          (Left              : in String;
                                 Right             : in String;
                                 Result            : out Boolean) is
    begin
        result := Is_Equal        (Left,Right);
    end Is_Equal;

    procedure Is_Equal          (Left              : in Substring;
                                 Right             : in String;
                                 Result            : out Boolean) is
    begin
        result := Is_Equal        (Left,Right);
    end Is_Equal;

    procedure Is_Equal          (Left              : in String;
                                 Right             : in Substring;
                                 Result            : out Boolean) is
    begin
        result := Is_Equal        (Left,Right);
    end Is_Equal;

    procedure Is_Less_Than      (Left              : in String;
                                 Right             : in String;
                                 Result            : out Boolean) is
    begin
        result := Is_Less_Than    (Left,Right);
    end Is_Less_Than;

    procedure Is_Less_Than      (Left              : in Substring;
                                 Right             : in String;
                                 Result            : out Boolean) is
    begin
        result := Is_Less_Than    (Left,Right);
    end Is_Less_Than;

    procedure Is_Less_Than      (Left              : in String;
                                 Right             : in Substring;
                                 Result            : out Boolean) is
    begin
        result := Is_Less_Than    (Left,Right);
    end Is_Less_Than;

    procedure Is_Greater_Than (Left                : in String;
                               Right               : in String;
                               Result              : out Boolean) is
    begin
        result := Is_Greater_Than (Left,Right);
    end Is_Greater_Than;

    procedure Is_Greater_Than (Left                : in Substring;
                               Right               : in String;
                               Result              : out Boolean) is
    begin
        result := Is_Greater_Than (Left,Right);
    end Is_Greater_Than;

    procedure Is_Greater_Than (Left                : in String;
                               Right               : in Substring;
                               Result              : out Boolean) is
    begin
        result := Is_Greater_Than (Left,Right);
    end Is_Greater_Than;

    procedure Length_Of         (The_String        : in String;
                                 Result            : out Natural) is
    begin
        result := Length_Of       (The_String);
    end Length_Of;

    procedure Is_Null           (The_String        : in String;
                                 Result            : out Boolean) is
    begin
        result := Is_Null         (The_String);
    end Is_Null;

    procedure Item_Of           (The_String        : in String;
                                 At_The_Position   : in Positive;
                                 Result            : out Item) is
    begin
        result := Item_Of         (The_String,At_The_Position);
    end Item_Of;

    procedure Substring_Of      (The_String        : in String;
                                 Result            : out Substring) is
    begin
        result := Substring_Of    (The_String);
    end Substring_Of;

    procedure Substring_Of      (The_String        : in String;
                                 From_The_Position : in Positive;
                                 To_The_Position   : in Positive;
                                 Result            : out Substring) is
    begin
        result :=
Substring_Of(The_String,From_The_Position,To_The_Position);
    end Substring_Of;

--  end of modification

    function Is_Equal (Left  : in String;
                       Right : in String) return Boolean is
    begin
        if Left.The_Length /= Right.The_Length then
            return False;
        else
            for Index in 1 .. Left.The_Length loop
                if Left.The_Items(Index) /= Right.The_Items(Index)
then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Is_Equal (Left  : in Substring;
                       Right : in String) return Boolean is
    begin
        if Left'Length /= Right.The_Length then
            return False;
        else
            for Index in 1 .. Left'Length loop
                if Left(Left'First + Index - 1) /=
Right.The_Items(Index) then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Is_Equal (Left  : in String;
                       Right : in Substring) return Boolean is
    begin
        if Left.The_Length /= Right'Length then
            return False;
        else
            for Index in 1 .. Left.The_Length loop
                if Left.The_Items(Index) /= Right(Right'First + Index
- 1) then
                    return False;
                end if;
            end loop;
            return True;
        end if;
    end Is_Equal;

    function Is_Less_Than (Left  : in String;
                           Right : in String) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right.The_Length then
                return False;
            elsif Left.The_Items(Index) < Right.The_Items(Index) then
                return True;
            elsif Right.The_Items(Index) < Left.The_Items(Index) then
                return False;
            end if;
        end loop;
        return (Left.The_Length < Right.The_Length);
    end Is_Less_Than;

    function Is_Less_Than (Left  : in Substring;
                           Right : in String) return Boolean is
    begin
        for Index in 1 .. Left'Length loop
            if Index > Right.The_Length then
                return False;
            elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
                return True;
            elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
```

```
            return False;
        end if;
    end loop;
    return (Left'Length < Right.The_Length);
end Is_Less_Than;

function Is_Less_Than (Left  : in String;
                       Right : in Substring) return Boolean is
begin
    for Index in 1 .. Left.The_Length loop
        if Index > Right'Length then
            return False;
        elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
            return True;
        elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
            return False;
        end if;
    end loop;
    return (Left.The_Length < Right'Length);
end Is_Less_Than;

function Is_Greater_Than (Left  : in String;
                          Right : in String) return Boolean is
begin
    for Index in 1 .. Left.The_Length loop
        if Index > Right.The_Length then
            return True;
        elsif Left.The_Items(Index) < Right.The_Items(Index) then
            return False;
        elsif Right.The_Items(Index) < Left.The_Items(Index) then
            return True;
        end if;
    end loop;
    return False;
end Is_Greater_Than;

function Is_Greater_Than (Left  : in Substring;
                          Right : in String) return Boolean is
begin
    for Index in 1 .. Left'Length loop
        if Index > Right.The_Length then
            return True;
        elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
            return False;
        elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
            return True;
        end if;
    end loop;
    return False;
end Is_Greater_Than;

function Is_Greater_Than (Left  : in String;
                          Right : in Substring) return Boolean is
begin
    for Index in 1 .. Left.The_Length loop
        if Index > Right'Length then
            return True;
        elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
            return False;
        elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
            return True;
        end if;
    end loop;
    return False;
end Is_Greater_Than;

function Length_Of (The_String : in String) return Natural is
begin
    return The_String.The_Length;
end Length_Of;

function Is_Null (The_String : in String) return Boolean is
begin
    return (The_String.The_Length = 0);
end Is_Null;

function Item_Of (The_String      : in String;
                  At_The_Position : in Positive) return Item is
begin
    if At_The_Position > The_String.The_Length then
        raise Position_Error;
    else
        return The_String.The_Items(At_The_Position);
    end if;
end Item_Of;

function Substring_Of (The_String : in String) return Substring is
    Temporary_Structure : Substring(1 .. 1);
begin
    return The_String.The_Items(1 .. The_String.The_Length);
exception
    when Constraint_Error =>
        return Temporary_Structure(1 .. 0);
end Substring_Of;

function Substring_Of (The_String        : in String;
                       From_The_Position : in Positive;
                       To_The_Position   : in Positive) return
Substring is
begin
    if (From_The_Position > The_String.The_Length) or else
       (To_The_Position > The_String.The_Length) or else
       (From_The_Position > To_The_Position) then
        raise Position_Error;
    else
        return The_String.The_Items(From_The_Position ..
To_The_Position);
    end if;
end Substring_Of;

procedure Iterate (Over_The_String : in String) is
    Continue : Boolean;
begin
    for The_Iterator in 1 .. Over_The_String.The_Length loop
        Process(Over_The_String.The_Items(The_Iterator),
Continue);
        exit when not Continue;
    end loop;
end Iterate;

end String_Sequential_Unbounded_Controlled_Iterator;
```

STRING SEQUENTIAL UNBOUNDED CONTROLLED ITERATOR

PSDL

```
TYPE String_Sequential_Unbounded_Controlled_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Substring : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive],
    func_"<" : FUNCTION[Left : Item, Right : Item, RETURN : Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_String : String
    OUTPUT
      The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_String : String,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Delete
  SPECIFICATION
    INPUT
      In_The_String : String,
      From_The_Position : Positive,
      To_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_String : String
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Substring : Substring
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Item : Item
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Substring,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : Substring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Less_Than
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END
```

```
OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Length_Of
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Natural
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Null
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Item_Of
SPECIFICATION
  INPUT
    The_String : String,
    At_The_Position : Positive
  OUTPUT
    Result : Item
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Substring_Of
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Substring
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Substring_Of
SPECIFICATION
  INPUT
    The_String : String,
    From_The_Position : Positive,
    To_The_Position : Positive
  OUTPUT
    Result : Substring
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Iterate
SPECIFICATION
  GENERIC
    Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
  INPUT
    Over_The_String : String
  EXCEPTIONS
    Overflow, Position_Error
END

END
IMPLEMENTATION ADA String_Sequential_Unbounded_Controlled_Iterator
END
```

STRING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Substring is array(Positive range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
package String_Sequential_Unbounded_Managed_Iterator is

    type String is limited private;

    procedure Copy     (From_The_String    : in     String;
                        To_The_String      : in out String);
    procedure Copy     (From_The_Substring : in     Substring;
                        To_The_String      : in out String);
    procedure Clear    (The_String         : in out String);
    procedure Prepend  (The_String         : in     String;
                        To_The_String      : in out String);
    procedure Prepend  (The_Substring      : in     Substring;
                        To_The_String      : in out String);
    procedure Append   (The_String         : in     String;
                        To_The_String      : in out String);
    procedure Append   (The_Substring      : in     Substring;
                        To_The_String      : in out String);
    procedure Insert   (The_String         : in     String;
                        In_The_String      : in out String;
                        At_The_Position    : in     Positive);
    procedure Insert   (The_Substring      : in     Substring;
                        In_The_String      : in out String;
                        At_The_Position    : in     Positive);
    procedure Delete   (In_The_String      : in out String;
                        From_The_Position  : in     Positive;
                        To_The_Position    : in     Positive);
    procedure Replace  (In_The_String      : in out String;
                        At_The_Position    : in     Positive;
                        With_The_String    : in     String);
    procedure Replace  (In_The_String      : in out String;
                        At_The_Position    : in     Positive;
                        With_The_Substring : in     Substring);
    procedure Set_Item (In_The_String      : in out String;
                        At_The_Position    : in     Positive;
                        With_The_Item      : in     Item);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal        (Left           : in String;
                               Right          : in String;
                               Result         : out Boolean);
    procedure Is_Equal        (Left           : in Substring;
                               Right          : in String;
                               Result         : out Boolean);
    procedure Is_Equal        (Left           : in String;
                               Right          : in Substring;
                               Result         : out Boolean);
    procedure Is_Less_Than    (Left           : in String;
                               Right          : in String;
                               Result         : out Boolean);
    procedure Is_Less_Than    (Left           : in Substring;
                               Right          : in String;
                               Result         : out Boolean);
    procedure Is_Less_Than    (Left           : in String;
                               Right          : in Substring;
                               Result         : out Boolean);
    procedure Is_Greater_Than (Left           : in String;
                               Right          : in String;
                               Result         : out Boolean);
    procedure Is_Greater_Than (Left           : in Substring;
                               Right          : in String;
                               Result         : out Boolean);
    procedure Is_Greater_Than (Left           : in String;
                               Right          : in Substring;
                               Result         : out Boolean);
    procedure Length_Of       (The_String        : in String;
                               Result            : out Natural);
    procedure Is_Null         (The_String        : in String;
                               Result            : out Boolean);
    procedure Item_Of         (The_String        : in String;
                               At_The_Position   : in Positive;
                               Result            : out Item);
    procedure Substring_Of    (The_String        : in String;
                               Result            : out Substring);
    procedure Substring_Of    (The_String        : in String;
                               From_The_Position : in Positive;
                               To_The_Position   : in Positive;
                               Result            : out Substring);

--  end of modification

    function Is_Equal        (Left              : in String;
                              Right             : in String)    return
Boolean;
    function Is_Equal        (Left              : in Substring;
                              Right             : in String)    return
Boolean;
    function Is_Equal        (Left              : in String;
                              Right             : in Substring) return
Boolean;
    function Is_Less_Than    (Left              : in String;
                              Right             : in String)    return
Boolean;
    function Is_Less_Than    (Left              : in Substring;
                              Right             : in String)    return
Boolean;
    function Is_Less_Than    (Left              : in String;
                              Right             : in Substring) return
Boolean;
    function Is_Greater_Than (Left              : in String;
                              Right             : in String)    return
Boolean;
    function Is_Greater_Than (Left              : in Substring;
                              Right             : in String)    return
Boolean;
    function Is_Greater_Than (Left              : in String;
                              Right             : in Substring) return
Boolean;
    function Length_Of       (The_String        : in String)    return
Natural;
    function Is_Null         (The_String        : in String)    return
Boolean;
    function Item_Of         (The_String        : in String;
                              At_The_Position   : in Positive) return
Item;
    function Substring_Of    (The_String        : in String)  return
Substring;
    function Substring_Of    (The_String        : in String;
                              From_The_Position : in Positive;
                              To_The_Position   : in Positive)return
Substring;

    generic
        with procedure Process (The_Item : in  Item;
                               Continue : out Boolean);
    procedure Iterate (Over_The_String : in String);

    Overflow       : exception;
    Position_Error : exception;

private
    type Structure is access Substring;
    type String is
        record
            The_Length : Natural := 0;
            The_Items  : Structure;
        end record;
end String_Sequential_Unbounded_Managed_Iterator;
```

STRING SEQUENTIAL UNBOUNDED MANAGED ITERATOR

PSDL

```
TYPE String_Sequential_Unbounded_Managed_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Substring : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive],
    func_"<" : FUNCTION[Left : Item, Right : Item, RETURN : Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_String : String
    OUTPUT
      The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_String : String,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Delete
  SPECIFICATION
    INPUT
      In_The_String : String,
      From_The_Position : Positive,
      To_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_String : String
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Substring : Substring
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Item : Item
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Substring,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : Substring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Less_Than
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END
```

```
OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Length_Of
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Natural
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Null
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Item_Of
SPECIFICATION
  INPUT
    The_String : String,
    At_The_Position : Positive
  OUTPUT
    Result : Item
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Substring_Of
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Substring
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Substring_Of
SPECIFICATION
  INPUT
    The_String : String,
    From_The_Position : Positive,
    To_The_Position : Positive
  OUTPUT
    Result : Substring
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Iterate
SPECIFICATION
  GENERIC
    Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
  INPUT
    Over_The_String : String
  EXCEPTIONS
    Overflow, Position_Error
END

END
IMPLEMENTATION ADA String_Sequential_Unbounded_Managed_Iterator
END
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Substring is array(Positive range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
package String_Sequential_Unbounded_Unmanaged_Noniterator is

    type String is limited private;

    procedure Copy     (From_The_String     : in     String;
                         To_The_String      : in out String);
    procedure Copy     (From_The_Substring  : in     Substring;
                         To_The_String      : in out String);
    procedure Clear    (The_String          : in out String);
    procedure Prepend  (The_String          : in     String;
                         To_The_String      : in out String);
    procedure Prepend  (The_Substring       : in     Substring;
                         To_The_String      : in out String);
    procedure Append   (The_String          : in     String;
                         To_The_String      : in out String);
    procedure Append   (The_Substring       : in     Substring;
                         To_The_String      : in out String);
    procedure Insert   (The_String          : in     String;
                         In_The_String      : in out String;
                         At_The_Position    : in     Positive);
    procedure Insert   (The_Substring       : in     Substring;
                         In_The_String      : in out String;
                         At_The_Position    : in     Positive);
    procedure Delete   (In_The_String       : in out String;
                         From_The_Position  : in     Positive;
                         To_The_Position    : in     Positive);
    procedure Replace  (In_The_String       : in out String;
                         At_The_Position    : in     Positive;
                         With_The_String    : in     String);
    procedure Replace  (In_The_String       : in out String;
                         At_The_Position    : in     Positive;
                         With_The_Substring : in     Substring);
    procedure Set_Item (In_The_String       : in out String;
                         At_The_Position    : in     Positive;
                         With_The_Item      : in     Item);


--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal          (Left              : in String;
                                  Right            : in String;
                                  Result           : out Boolean);
    procedure Is_Equal          (Left              : in Substring;
                                  Right            : in String;
                                  Result           : out Boolean);
    procedure Is_Equal          (Left              : in String;
                                  Right            : in Substring;
                                  Result           : out Boolean);
    procedure Is_Less_Than      (Left              : in String;
                                  Right            : in String;
                                  Result           : out Boolean);
    procedure Is_Less_Than      (Left              : in Substring;
                                  Right            : in String;
                                  Result           : out Boolean);
    procedure Is_Less_Than      (Left              : in String;
                                  Right            : in Substring;
                                  Result           : out Boolean);
    procedure Is_Greater_Than   (Left              : in String;
                                  Right            : in String;
                                  Result           : out Boolean);
    procedure Is_Greater_Than   (Left              : in Substring;
                                  Right            : in String;
                                  Result           : out Boolean);
    procedure Is_Greater_Than   (Left              : in String;
                                  Right            : in Substring;
                                  Result           : out Boolean);
    procedure Length_Of         (The_String        : in String;
                                  Result           : out Natural);
    procedure Is_Null           (The_String        : in String;
                                  Result           : out Boolean);
    procedure Item_Of           (The_String        : in String;
                                  At_The_Position  : in Positive;
                                  Result           : out Item);
    procedure Substring_Of      (The_String        : in String;
                                  Result           : out Substring);
    procedure Substring_Of      (The_String        : in String;
                                  From_The_Position : in Positive;
                                  To_The_Position  : in Positive;
                                  Result           : out Substring);

--  end of modification

    function Is_Equal          (Left              : in String;
                                 Right            : in String)    return
Boolean;
    function Is_Equal          (Left              : in Substring;
                                 Right            : in String)    return
Boolean;
    function Is_Equal          (Left              : in String;
                                 Right            : in Substring) return
Boolean;
    function Is_Less_Than      (Left              : in String;
                                 Right            : in String)    return
Boolean;
    function Is_Less_Than      (Left              : in Substring;
                                 Right            : in String)    return
Boolean;
    function Is_Less_Than      (Left              : in String;
                                 Right            : in Substring) return
Boolean;
    function Is_Greater_Than   (Left              : in String;
                                 Right            : in String)    return
Boolean;
    function Is_Greater_Than   (Left              : in Substring;
                                 Right            : in String)    return
Boolean;
    function Is_Greater_Than   (Left              : in String;
                                 Right            : in Substring) return
Boolean;
    function Length_Of         (The_String        : in String)    return
Natural;
    function Is_Null           (The_String        : in String)    return
Boolean;
    function Item_Of           (The_String        : in String;
                                 At_The_Position  : in Positive)  return
Item;
    function Substring_Of      (The_String        : in String)  return
Substring;
    function Substring_Of      (The_String        : in String;
                                 From_The_Position : in Positive;
                                 To_The_Position  : in Positive)return
Substring;

    Overflow       : exception;
    Position_Error : exception;

private
    type Structure is access Substring;
    type String is
        record
            The_Length : Natural := 0;
            The_Items  : Structure;
        end record;
end String_Sequential_Unbounded_Unmanaged_Noniterator;
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body String_Sequential_Unbounded_Unmanaged_Noniterator is

    procedure Set (The_String         : in out String;
                   To_The_Size        : in     Natural;
                   Preserve_The_Value : in     Boolean) is
        Temporary_Structure : Structure;
    begin
        if To_The_Size = 0 then
            The_String.The_Items := null;
        elsif The_String.The_Items = null then
            The_String.The_Items := new Substring(1 .. To_The_Size);
        elsif To_The_Size > The_String.The_Items'Length then
            if Preserve_The_Value then
                Temporary_Structure := new Substring(1 ..
To_The_Size);
                Temporary_Structure(1 .. The_String.The_Length) :=
                  The_String.The_Items(1 .. The_String.The_Length);
                The_String.The_Items := Temporary_Structure;
            else
                The_String.The_Items := new Substring(1 ..
To_The_Size);
            end if;
        end if;
        The_String.The_Length := To_The_Size;
    end Set;

    procedure Copy (From_The_String : in     String;
                    To_The_String   : in out String) is
    begin
        Set(To_The_String,
            To_The_Size        => From_The_String.The_Length,
            Preserve_The_Value => False);
        To_The_String.The_Items(1 .. From_The_String.The_Length) :=
          From_The_String.The_Items(1 .. From_The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Copy (From_The_Substring : in     Substring;
                    To_The_String      : in out String) is
    begin
        Set(To_The_String,
            To_The_Size        => From_The_Substring'Length,
            Preserve_The_Value => False);
        To_The_String.The_Items(1 .. From_The_Substring'Length) :=
          From_The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_String : in out String) is
    begin
        Set(The_String,
            To_The_Size        => 0,
            Preserve_The_Value => False);
    end Clear;

    procedure Prepend (The_String    : in     String;
                       To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_String.The_Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((The_String.The_Length + 1) ..
New_Length)
          := To_The_String.The_Items(1 .. Old_Length);
        To_The_String.The_Items(1 .. The_String.The_Length) :=
          The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Prepend;

    procedure Prepend (The_Substring : in     Substring;
                       To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_Substring'Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((The_Substring'Length + 1) ..
New_Length)
          := To_The_String.The_Items(1 .. Old_Length);
        To_The_String.The_Items(1 .. The_Substring'Length) :=
The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Prepend;

    procedure Append (The_String    : in     String;
                      To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_String.The_Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((Old_Length + 1) .. New_Length)
          := The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Append;

    procedure Append (The_Substring : in     Substring;
                      To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                       To_The_String.The_Length + The_Substring'Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((Old_Length + 1) .. New_Length)
          := The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Append;

    procedure Insert (The_String      : in     String;
                      In_The_String   : in out String;
                      At_The_Position : in     Positive) is
        Old_Length   : Natural := In_The_String.The_Length;
        New_Length   : Natural :=
                         In_The_String.The_Length +
The_String.The_Length;
        End_Position : Natural :=
                         At_The_Position + The_String.The_Length;
    begin
        if At_The_Position > In_The_String.The_Length then
            raise Position_Error;
        else
            Set(In_The_String,
                To_The_Size        => New_Length,
                Preserve_The_Value => True);
            In_The_String.The_Items(End_Position .. New_Length) :=
              In_The_String.The_Items(At_The_Position .. Old_Length);
            In_The_String.The_Items(At_The_Position .. (End_Position -
1)) :=
              The_String.The_Items(1 .. The_String.The_Length);
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Insert (The_Substring   : in     Substring;
                      In_The_String   : in out String;
                      At_The_Position : in     Positive) is
        Old_Length   : Natural := In_The_String.The_Length;
        New_Length   : Natural :=
                         In_The_String.The_Length +
The_Substring'Length;
        End_Position : Natural :=
                         At_The_Position + The_Substring'Length;
    begin
        if At_The_Position > In_The_String.The_Length then
            raise Position_Error;
        else
            Set(In_The_String,
                To_The_Size        => New_Length,
                Preserve_The_Value => True);
            In_The_String.The_Items(End_Position .. New_Length) :=
              In_The_String.The_Items(At_The_Position .. Old_Length);
            In_The_String.The_Items(At_The_Position .. (End_Position -
1)) :=
              The_Substring;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Delete (In_The_String      : in out String;
```

```
                    From_The_Position : in     Positive;
                    To_The_Position   : in     Positive) is
      New_Length : Natural;
   begin
      if (From_The_Position > In_The_String.The_Length) or else
         (To_The_Position > In_The_String.The_Length) or else
         (From_The_Position > To_The_Position) then
          raise Position_Error;
      else
          New_Length := In_The_String.The_Length -
                        (To_The_Position - From_The_Position + 1);
          In_The_String.The_Items(From_The_Position .. New_Length)
:=
            In_The_String.The_Items
            ((To_The_Position + 1) .. In_The_String.The_Length);
          Set(In_The_String,
              To_The_Size        => New_Length,
              Preserve_The_Value => True);
      end if;
   end Delete;

   procedure Replace (In_The_String   : in out String;
                      At_The_Position : in     Positive;
                      With_The_String : in     String) is
      End_Position : Natural :=
                      At_The_Position + With_The_String.The_Length -
1;
   begin
      if (At_The_Position > In_The_String.The_Length) or else
         (End_Position > In_The_String.The_Length) then
          raise Position_Error;
      else
          In_The_String.The_Items(At_The_Position .. End_Position)
:=
            With_The_String.The_Items(1 ..
With_The_String.The_Length);
      end if;
   end Replace;

   procedure Replace (In_The_String      : in out String;
                      At_The_Position    : in     Positive;
                      With_The_Substring : in     Substring) is
      End_Position : Natural :=
                      At_The_Position + With_The_Substring'Length -
1;
   begin
      if (At_The_Position > In_The_String.The_Length) or else
         (End_Position > In_The_String.The_Length) then
          raise Position_Error;
      else
          In_The_String.The_Items(At_The_Position .. End_Position)
:=
            With_The_Substring;
      end if;
   end Replace;

   procedure Set_Item (In_The_String   : in out String;
                       At_The_Position : in     Positive;
                       With_The_Item   : in     Item) is
   begin
      if At_The_Position > In_The_String.The_Length then
          raise Position_Error;
      else
          In_The_String.The_Items(At_The_Position) := With_The_Item;
      end if;
   end Set_Item;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

   procedure Is_Equal          (Left               : in String;
                                Right              : in String;
                                Result             : out Boolean) is
   begin
      result := Is_Equal         (Left,Right);
   end Is_Equal;

   procedure Is_Equal          (Left               : in Substring;
                                Right              : in String;
                                Result             : out Boolean) is
   begin
      result := Is_Equal         (Left,Right);
   end Is_Equal;

   procedure Is_Equal          (Left               : in String;
                                Right              : in Substring;
                                Result             : out Boolean) is
   begin
      result := Is_Equal         (Left,Right);
   end Is_Equal;

   procedure Is_Less_Than      (Left               : in String;
                                Right              : in String;
                                Result             : out Boolean) is
   begin
      result := Is_Less_Than     (Left,Right);
   end Is_Less_Than;

   procedure Is_Less_Than      (Left               : in Substring;
                                Right              : in String;
                                Result             : out Boolean) is
   begin
      result := Is_Less_Than     (Left,Right);
   end Is_Less_Than;

   procedure Is_Less_Than      (Left               : in String;
                                Right              : in Substring;
                                Result             : out Boolean) is
   begin
      result := Is_Less_Than     (Left,Right);
   end Is_Less_Than;

   procedure Is_Greater_Than (Left                 : in String;
                              Right                : in String;
                              Result               : out Boolean) is
   begin
      result := Is_Greater_Than (Left,Right);
   end Is_Greater_Than;

   procedure Is_Greater_Than (Left                 : in Substring;
                              Right                : in String;
                              Result               : out Boolean) is
   begin
      result := Is_Greater_Than (Left,Right);
   end Is_Greater_Than;

   procedure Is_Greater_Than (Left                 : in String;
                              Right                : in Substring;
                              Result               : out Boolean) is
   begin
      result := Is_Greater_Than (Left,Right);
   end Is_Greater_Than;

   procedure Length_Of       (The_String           : in String;
                              Result               : out Natural) is
   begin
      result := Length_Of        (The_String);
   end Length_Of;

   procedure Is_Null         (The_String           : in String;
                              Result               : out Boolean) is
   begin
      result := Is_Null          (The_String);
   end Is_Null;

   procedure Item_Of         (The_String           : in String;
                              At_The_Position      : in Positive;
                              Result               : out Item) is
   begin
      result := Item_Of          (The_String,At_The_Position);
   end Item_Of;

   procedure Substring_Of    (The_String           : in String;
                              Result               : out Substring) is
   begin
      result := Substring_Of     (The_String);
   end Substring_Of;

   procedure Substring_Of    (The_String           : in String;
                              From_The_Position    : in Positive;
                              To_The_Position      : in Positive;
                              Result               : out Substring) is
   begin
      result :=
Substring_Of(The_String,From_The_Position,To_The_Position);
   end Substring_Of;

--  end of modification

   function Is_Equal (Left  : in String;
                      Right : in String) return Boolean is
   begin
      if Left.The_Length /= Right.The_Length then
          return False;
      else
          for Index in 1 .. Left.The_Length loop
              if Left.The_Items(Index) /= Right.The_Items(Index)
then
                  return False;
              end if;
          end loop;
          return True;
      end if;
   end Is_Equal;

   function Is_Equal (Left  : in Substring;
                      Right : in String) return Boolean is
   begin
      if Left'Length /= Right.The_Length then
          return False;
      else
          for Index in 1 .. Left'Length loop
              if Left(Left'First + Index - 1) /=
Right.The_Items(Index) then
                  return False;
              end if;
          end loop;
          return True;
      end if;
   end Is_Equal;

   function Is_Equal (Left  : in String;
                      Right : in Substring) return Boolean is
   begin
      if Left.The_Length /= Right'Length then
          return False;
      else
          for Index in 1 .. Left.The_Length loop
              if Left.The_Items(Index) /= Right(Right'First + Index
- 1) then
                  return False;
              end if;
          end loop;
          return True;
      end if;
```

```
    end Is_Equal;

    function Is_Less_Than (Left  : in String;
                           Right : in String) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right.The_Length then
                return False;
            elsif Left.The_Items(Index) < Right.The_Items(Index) then
                return True;
            elsif Right.The_Items(Index) < Left.The_Items(Index) then
                return False;
            end if;
        end loop;
        return (Left.The_Length < Right.The_Length);
    end Is_Less_Than;

    function Is_Less_Than (Left  : in Substring;
                           Right : in String) return Boolean is
    begin
        for Index in 1 .. Left'Length loop
            if Index > Right.The_Length then
                return False;
            elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
                return True;
            elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
                return False;
            end if;
        end loop;
        return (Left'Length < Right.The_Length);
    end Is_Less_Than;

    function Is_Less_Than (Left  : in String;
                           Right : in Substring) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right'Length then
                return False;
            elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
                return True;
            elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
                return False;
            end if;
        end loop;
        return (Left.The_Length < Right'Length);
    end Is_Less_Than;

    function Is_Greater_Than (Left  : in String;
                              Right : in String) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right.The_Length then
                return True;
            elsif Left.The_Items(Index) < Right.The_Items(Index) then
                return False;
            elsif Right.The_Items(Index) < Left.The_Items(Index) then
                return True;
            end if;
        end loop;
        return False;
    end Is_Greater_Than;

    function Is_Greater_Than (Left  : in Substring;
                              Right : in String) return Boolean is
    begin
        for Index in 1 .. Left'Length loop
            if Index > Right.The_Length then
                return True;
            elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
                return False;
            elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
                return True;
            end if;
        end loop;
        return False;
    end Is_Greater_Than;

    function Is_Greater_Than (Left  : in String;
                              Right : in Substring) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right'Length then
                return True;
            elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
                return False;
            elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
                return True;
            end if;
        end loop;
        return False;
    end Is_Greater_Than;

    function Length_Of (The_String : in String) return Natural is
    begin
        return The_String.The_Length;
    end Length_Of;

    function Is_Null (The_String : in String) return Boolean is
    begin
        return (The_String.The_Length = 0);
    end Is_Null;

    function Item_Of (The_String      : in String;
                      At_The_Position : in Positive) return Item is
    begin
        if At_The_Position > The_String.The_Length then
            raise Position_Error;
        else
            return The_String.The_Items(At_The_Position);
        end if;
    end Item_Of;

    function Substring_Of (The_String : in String) return Substring is
        Temporary_Structure : Substring(1 .. 1);
    begin
        return The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Constraint_Error =>
            return Temporary_Structure(1 .. 0);
    end Substring_Of;

    function Substring_Of (The_String        : in String;
                           From_The_Position : in Positive;
                           To_The_Position   : in Positive) return
Substring is
    begin
        if (From_The_Position > The_String.The_Length) or else
           (To_The_Position > The_String.The_Length) or else
           (From_The_Position > To_The_Position) then
            raise Position_Error;
        else
            return The_String.The_Items(From_The_Position ..
To_The_Position);
        end if;
    end Substring_Of;

end String_Sequential_Unbounded_Unmanaged_Noniterator;
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED NONITERATOR

PSDL

```
TYPE String_Sequential_Unbounded_Unmanaged_Noniterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Substring : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive],
    func_"<" : FUNCTION[Left : Item, Right : Item, RETURN : Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_String : String
    OUTPUT
      The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_String : String,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Delete
  SPECIFICATION
    INPUT
      In_The_String : String,
      From_The_Position : Positive,
      To_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_String : String
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Substring : Substring
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Item : Item
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Substring,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : Substring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Less_Than
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END
```

```
OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Length_Of
  SPECIFICATION
    INPUT
      The_String : String
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_String : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Item_Of
  SPECIFICATION
    INPUT
      The_String : String,
      At_The_Position : Positive
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Substring_Of
  SPECIFICATION
    INPUT
      The_String : String
    OUTPUT
      Result : Substring
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Substring_Of
  SPECIFICATION
    INPUT
      The_String : String,
      From_The_Position : Positive,
      To_The_Position : Positive
    OUTPUT
      Result : Substring
    EXCEPTIONS
      Overflow, Position_Error
  END

END
IMPLEMENTATION ADA String_Sequential_Unbounded_Unmanaged_Noniterator
END
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA SPECIFICATIONS

```
generic
    type Item is private;
    type Substring is array(Positive range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
package String_Sequential_Unbounded_Unmanaged_Iterator is

    type String is limited private;

    procedure Copy    (From_The_String    : in     String;
                       To_The_String      : in out String);
    procedure Copy    (From_The_Substring : in     Substring;
                       To_The_String      : in out String);
    procedure Clear   (The_String         : in out String);
    procedure Prepend (The_String         : in     String;
                       To_The_String      : in out String);
    procedure Prepend (The_Substring      : in     Substring;
                       To_The_String      : in out String);
    procedure Append  (The_String         : in     String;
                       To_The_String      : in out String);
    procedure Append  (The_Substring      : in     Substring;
                       To_The_String      : in out String);
    procedure Insert  (The_String         : in     String;
                       In_The_String      : in out String;
                       At_The_Position    : in     Positive);
    procedure Insert  (The_Substring      : in     Substring;
                       In_The_String      : in out String;
                       At_The_Position    : in     Positive);
    procedure Delete  (In_The_String      : in out String;
                       From_The_Position  : in     Positive;
                       To_The_Position    : in     Positive);
    procedure Replace (In_The_String      : in out String;
                       At_The_Position    : in     Positive;
                       With_The_String    : in     String);
    procedure Replace (In_The_String      : in out String;
                       At_The_Position    : in     Positive;
                       With_The_Substring : in     Substring);
    procedure Set_Item (In_The_String     : in out String;
                       At_The_Position    : in     Positive;
                       With_The_Item      : in     Item);

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal          (Left             : in String;
                                  Right           : in String;
                                  Result          : out Boolean);
    procedure Is_Equal          (Left             : in Substring;
                                  Right           : in String;
                                  Result          : out Boolean);
    procedure Is_Equal          (Left             : in String;
                                  Right           : in Substring;
                                  Result          : out Boolean);
    procedure Is_Less_Than      (Left             : in String;
                                  Right           : in String;
                                  Result          : out Boolean);
    procedure Is_Less_Than      (Left             : in Substring;
                                  Right           : in String;
                                  Result          : out Boolean);
    procedure Is_Less_Than      (Left             : in String;
                                  Right           : in Substring;
                                  Result          : out Boolean);
    procedure Is_Greater_Than   (Left             : in String;
                                  Right           : in String;
                                  Result          : out Boolean);
    procedure Is_Greater_Than   (Left             : in Substring;
                                  Right           : in String;
                                  Result          : out Boolean);
    procedure Is_Greater_Than   (Left             : in String;
                                  Right           : in Substring;
                                  Result          : out Boolean);
    procedure Length_Of         (The_String       : in String;
                                  Result          : out Natural);
    procedure Is_Null           (The_String       : in String;
                                  Result          : out Boolean);
    procedure Item_Of           (The_String       : in String;
                                  At_The_Position : in Positive;
                                  Result          : out Item);
    procedure Substring_Of      (The_String       : in String;
                                  Result          : out Substring);
    procedure Substring_Of      (The_String       : in String;
                                  From_The_Position : in Positive;
                                  To_The_Position : in Positive;
                                  Result          : out Substring);

--  end of modification

    function Is_Equal        (Left            : in String;
                              Right           : in String)    return
Boolean;
    function Is_Equal        (Left            : in Substring;
                              Right           : in String)    return
Boolean;
    function Is_Equal        (Left            : in String;
                              Right           : in Substring) return
Boolean;
    function Is_Less_Than    (Left            : in String;
                              Right           : in String)    return
Boolean;
    function Is_Less_Than    (Left            : in Substring;
                              Right           : in String)    return
Boolean;
    function Is_Less_Than    (Left            : in String;
                              Right           : in Substring) return
Boolean;
    function Is_Greater_Than (Left            : in String;
                              Right           : in String)    return
Boolean;
    function Is_Greater_Than (Left            : in Substring;
                              Right           : in String)    return
Boolean;
    function Is_Greater_Than (Left            : in String;
                              Right           : in Substring) return
Boolean;
    function Length_Of       (The_String      : in String)    return
Natural;
    function Is_Null         (The_String      : in String)    return
Boolean;
    function Item_Of         (The_String      : in String;
                              At_The_Position : in Positive)  return
Item;
    function Substring_Of    (The_String      : in String)  return
Substring;
    function Substring_Of    (The_String      : in String;
                              From_The_Position : in Positive;
                              To_The_Position : in Positive)return
Substring;

    generic
        with procedure Process (The_Item : in  Item;
                                Continue : out Boolean);
    procedure Iterate (Over_The_String : in String);

    Overflow       : exception;
    Position_Error : exception;

private
    type Structure is access Substring;
    type String is
        record
            The_Length : Natural := 0;
            The_Items  : Structure;
        end record;
end String_Sequential_Unbounded_Unmanaged_Iterator;
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--            "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
package body String_Sequential_Unbounded_Unmanaged_Iterator is

    procedure Set (The_String        : in out String;
                   To_The_Size       : in     Natural;
                   Preserve_The_Value : in    Boolean) is
        Temporary_Structure : Structure;
    begin
        if To_The_Size = 0 then
            The_String.The_Items := null;
        elsif The_String.The_Items = null then
            The_String.The_Items := new Substring(1 .. To_The_Size);
        elsif To_The_Size > The_String.The_Items'Length then
            if Preserve_The_Value then
                Temporary_Structure := new Substring(1 ..
To_The_Size);
                Temporary_Structure(1 .. The_String.The_Length) :=
                  The_String.The_Items(1 .. The_String.The_Length);
                The_String.The_Items := Temporary_Structure;
            else
                The_String.The_Items := new Substring(1 ..
To_The_Size);
            end if;
        end if;
        The_String.The_Length := To_The_Size;
    end Set;

    procedure Copy (From_The_String : in     String;
                    To_The_String   : in out String) is
    begin
        Set(To_The_String,
            To_The_Size        => From_The_String.The_Length,
            Preserve_The_Value => False);
        To_The_String.The_Items(1 .. From_The_String.The_Length) :=
          From_The_String.The_Items(1 .. From_The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Copy (From_The_Substring : in     Substring;
                    To_The_String      : in out String) is
    begin
        Set(To_The_String,
            To_The_Size        => From_The_Substring'Length,
            Preserve_The_Value => False);
        To_The_String.The_Items(1 .. From_The_Substring'Length) :=
          From_The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Copy;

    procedure Clear (The_String : in out String) is
    begin
        Set(The_String,
            To_The_Size        => 0,
            Preserve_The_Value => False);
    end Clear;

    procedure Prepend (The_String    : in     String;
                       To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                     To_The_String.The_Length + The_String.The_Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((The_String.The_Length + 1) ..
New_Length)
          := To_The_String.The_Items(1 .. Old_Length);
        To_The_String.The_Items(1 .. The_String.The_Length) :=
          The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Prepend;

    procedure Prepend (The_Substring : in     Substring;
                       To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                     To_The_String.The_Length + The_Substring'Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((The_Substring'Length + 1) ..
New_Length)
          := To_The_String.The_Items(1 .. Old_Length);
        To_The_String.The_Items(1 .. The_Substring'Length) :=
The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Prepend;

    procedure Append (The_String    : in     String;
                      To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                     To_The_String.The_Length + The_String.The_Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((Old_Length + 1) .. New_Length)
          := The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Storage_Error =>
            raise Overflow;
    end Append;

    procedure Append (The_Substring : in     Substring;
                      To_The_String : in out String) is
        Old_Length : Natural := To_The_String.The_Length;
        New_Length : Natural :=
                     To_The_String.The_Length + The_Substring'Length;
    begin
        Set(To_The_String,
            To_The_Size        => New_Length,
            Preserve_The_Value => True);
        To_The_String.The_Items((Old_Length + 1) .. New_Length)
          := The_Substring;
    exception
        when Storage_Error =>
            raise Overflow;
    end Append;

    procedure Insert (The_String      : in     String;
                      In_The_String   : in out String;
                      At_The_Position : in     Positive) is
        Old_Length   : Natural := In_The_String.The_Length;
        New_Length   : Natural :=
                       In_The_String.The_Length +
The_String.The_Length;
        End_Position : Natural :=
                       At_The_Position + The_String.The_Length;
    begin
        if At_The_Position > In_The_String.The_Length then
            raise Position_Error;
        else
            Set(In_The_String,
                To_The_Size        => New_Length,
                Preserve_The_Value => True);
            In_The_String.The_Items(End_Position .. New_Length) :=
              In_The_String.The_Items(At_The_Position .. Old_Length);
            In_The_String.The_Items(At_The_Position .. (End_Position -
1)) :=
              The_String.The_Items(1 .. The_String.The_Length);
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Insert (The_Substring   : in     Substring;
                      In_The_String   : in out String;
                      At_The_Position : in     Positive) is
        Old_Length   : Natural := In_The_String.The_Length;
        New_Length   : Natural :=
                       In_The_String.The_Length +
The_Substring'Length;
        End_Position : Natural :=
                       At_The_Position + The_Substring'Length;
    begin
        if At_The_Position > In_The_String.The_Length then
            raise Position_Error;
        else
            Set(In_The_String,
                To_The_Size        => New_Length,
                Preserve_The_Value => True);
            In_The_String.The_Items(End_Position .. New_Length) :=
              In_The_String.The_Items(At_The_Position .. Old_Length);
            In_The_String.The_Items(At_The_Position .. (End_Position -
1)) :=
              The_Substring;
        end if;
    exception
        when Storage_Error =>
            raise Overflow;
    end Insert;

    procedure Delete (In_The_String     : in out String;
```

```
                      From_The_Position : in     Positive;
                      To_The_Position   : in     Positive) is
       New_Length : Natural;
    begin
       if (From_The_Position > In_The_String.The_Length) or else
          (To_The_Position > In_The_String.The_Length) or else
          (From_The_Position > To_The_Position) then
           raise Position_Error;
       else
           New_Length := In_The_String.The_Length -
                         (To_The_Position - From_The_Position + 1);
           In_The_String.The_Items(From_The_Position .. New_Length)
:=
             In_The_String.The_Items
             ((To_The_Position + 1) .. In_The_String.The_Length);
           Set(In_The_String,
               To_The_Size        => New_Length,
               Preserve_The_Value => True);
       end if;
    end Delete;

    procedure Replace (In_The_String   : in out String;
                       At_The_Position : in     Positive;
                       With_The_String : in     String) is
       End_Position : Natural :=
                       At_The_Position + With_The_String.The_Length -
1;
    begin
       if (At_The_Position > In_The_String.The_Length) or else
          (End_Position > In_The_String.The_Length) then
           raise Position_Error;
       else
           In_The_String.The_Items(At_The_Position .. End_Position)
:=
             With_The_String.The_Items(1 ..
With_The_String.The_Length);
       end if;
    end Replace;

    procedure Replace (In_The_String      : in out String;
                       At_The_Position    : in     Positive;
                       With_The_Substring : in     Substring) is
       End_Position : Natural :=
                       At_The_Position + With_The_Substring'Length -
1;
    begin
       if (At_The_Position > In_The_String.The_Length) or else
          (End_Position > In_The_String.The_Length) then
           raise Position_Error;
       else
           In_The_String.The_Items(At_The_Position .. End_Position)
:=
             With_The_Substring;
       end if;
    end Replace;

    procedure Set_Item (In_The_String   : in out String;
                        At_The_Position : in     Positive;
                        With_The_Item   : in     Item) is
    begin
       if At_The_Position > In_The_String.The_Length then
           raise Position_Error;
       else
           In_The_String.The_Items(At_The_Position) := With_The_Item;
       end if;
    end Set_Item;

--  modified by Vincent Hong and Tuan Nguyen
--  date:   9 April 1995
--  adding procedures to replace functions

    procedure Is_Equal          (Left             : in String;
                                 Right            : in String;
                                 Result           : out Boolean) is
    begin
       result := Is_Equal          (Left,Right);
    end Is_Equal;

    procedure Is_Equal          (Left             : in Substring;
                                 Right            : in String;
                                 Result           : out Boolean) is
    begin
       result := Is_Equal          (Left,Right);
    end Is_Equal;

    procedure Is_Equal          (Left             : in String;
                                 Right            : in Substring;
                                 Result           : out Boolean) is
    begin
       result := Is_Equal          (Left,Right);
    end Is_Equal;

    procedure Is_Less_Than      (Left             : in String;
                                 Right            : in String;
                                 Result           : out Boolean) is
    begin
       result := Is_Less_Than      (Left,Right);
    end Is_Less_Than;

    procedure Is_Less_Than      (Left             : in Substring;
                                 Right            : in String;
                                 Result           : out Boolean) is
    begin
       result := Is_Less_Than      (Left,Right);
    end Is_Less_Than;

    procedure Is_Less_Than      (Left             : in String;
                                 Right            : in Substring;
                                 Result           : out Boolean) is
    begin
       result := Is_Less_Than      (Left,Right);
    end Is_Less_Than;

    procedure Is_Greater_Than (Left               : in String;
                               Right              : in String;
                               Result             : out Boolean) is
    begin
       result := Is_Greater_Than (Left,Right);
    end Is_Greater_Than;

    procedure Is_Greater_Than (Left               : in Substring;
                               Right              : in String;
                               Result             : out Boolean) is
    begin
       result := Is_Greater_Than (Left,Right);
    end Is_Greater_Than;

    procedure Is_Greater_Than (Left               : in String;
                               Right              : in Substring;
                               Result             : out Boolean) is
    begin
       result := Is_Greater_Than (Left,Right);
    end Is_Greater_Than;

    procedure Length_Of        (The_String        : in String;
                                Result            : out Natural) is
    begin
       result := Length_Of        (The_String);
    end Length_Of;

    procedure Is_Null          (The_String        : in String;
                                Result            : out Boolean) is
    begin
       result := Is_Null          (The_String);
    end Is_Null;

    procedure Item_Of          (The_String        : in String;
                                At_The_Position   : in Positive;
                                Result            : out Item) is
    begin
       result := Item_Of          (The_String,At_The_Position);
    end Item_Of;

    procedure Substring_Of     (The_String        : in String;
                                Result            : out Substring) is
    begin
       result := Substring_Of     (The_String);
    end Substring_Of;

    procedure Substring_Of     (The_String        : in String;
                                From_The_Position : in Positive;
                                To_The_Position   : in Positive;
                                Result            : out Substring) is
    begin
       result :=
Substring_Of(The_String,From_The_Position,To_The_Position);
    end Substring_Of;

--  end of modification

    function Is_Equal (Left  : in String;
                       Right : in String) return Boolean is
    begin
       if Left.The_Length /= Right.The_Length then
           return False;
       else
           for Index in 1 .. Left.The_Length loop
               if Left.The_Items(Index) /= Right.The_Items(Index)
then
                   return False;
               end if;
           end loop;
           return True;
       end if;
    end Is_Equal;

    function Is_Equal (Left  : in Substring;
                       Right : in String) return Boolean is
    begin
       if Left'Length /= Right.The_Length then
           return False;
       else
           for Index in 1 .. Left'Length loop
               if Left(Left'First + Index - 1) /=
Right.The_Items(Index) then
                   return False;
               end if;
           end loop;
           return True;
       end if;
    end Is_Equal;

    function Is_Equal (Left  : in String;
                       Right : in Substring) return Boolean is
    begin
       if Left.The_Length /= Right'Length then
           return False;
       else
           for Index in 1 .. Left.The_Length loop
               if Left.The_Items(Index) /= Right(Right'First + Index
- 1) then
                   return False;
               end if;
           end loop;
           return True;
       end if;
```

```
    end Is_Equal;

    function Is_Less_Than (Left  : in String;
                           Right : in String) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right.The_Length then
                return False;
            elsif Left.The_Items(Index) < Right.The_Items(Index) then
                return True;
            elsif Right.The_Items(Index) < Left.The_Items(Index) then
                return False;
            end if;
        end loop;
        return (Left.The_Length < Right.The_Length);
    end Is_Less_Than;

    function Is_Less_Than (Left  : in Substring;
                           Right : in String) return Boolean is
    begin
        for Index in 1 .. Left'Length loop
            if Index > Right.The_Length then
                return False;
            elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
                return True;
            elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
                return False;
            end if;
        end loop;
        return (Left'Length < Right.The_Length);
    end Is_Less_Than;

    function Is_Less_Than (Left  : in String;
                           Right : in Substring) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right'Length then
                return False;
            elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
                return True;
            elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
                return False;
            end if;
        end loop;
        return (Left.The_Length < Right'Length);
    end Is_Less_Than;

    function Is_Greater_Than (Left  : in String;
                              Right : in String) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right.The_Length then
                return True;
            elsif Left.The_Items(Index) < Right.The_Items(Index) then
                return False;
            elsif Right.The_Items(Index) < Left.The_Items(Index) then
                return True;
            end if;
        end loop;
        return False;
    end Is_Greater_Than;

    function Is_Greater_Than (Left  : in Substring;
                              Right : in String) return Boolean is
    begin
        for Index in 1 .. Left'Length loop
            if Index > Right.The_Length then
                return True;
            elsif Left(Left'First + Index - 1) <
Right.The_Items(Index) then
                return False;
            elsif Right.The_Items(Index) < Left(Left'First + Index -
1) then
                return True;
            end if;
        end loop;
        return False;
    end Is_Greater_Than;

    function Is_Greater_Than (Left  : in String;
                              Right : in Substring) return Boolean is
    begin
        for Index in 1 .. Left.The_Length loop
            if Index > Right'Length then
                return True;
            elsif Left.The_Items(Index) < Right(Right'First + Index -
1) then
                return False;
            elsif Right(Right'First + Index - 1) <
Left.The_Items(Index) then
                return True;
            end if;
        end loop;
        return False;
    end Is_Greater_Than;

    function Length_Of (The_String : in String) return Natural is
    begin
        return The_String.The_Length;
    end Length_Of;

    function Is_Null (The_String : in String) return Boolean is
    begin
        return (The_String.The_Length = 0);
    end Is_Null;

    function Item_Of (The_String      : in String;
                      At_The_Position : in Positive) return Item is
    begin
        if At_The_Position > The_String.The_Length then
            raise Position_Error;
        else
            return The_String.The_Items(At_The_Position);
        end if;
    end Item_Of;

    function Substring_Of (The_String : in String) return Substring is
        Temporary_Structure : Substring(1 .. 1);
    begin
        return The_String.The_Items(1 .. The_String.The_Length);
    exception
        when Constraint_Error =>
            return Temporary_Structure(1 .. 0);
    end Substring_Of;

    function Substring_Of (The_String        : in String;
                           From_The_Position : in Positive;
                           To_The_Position   : in Positive) return
Substring is
    begin
        if (From_The_Position > The_String.The_Length) or else
           (To_The_Position > The_String.The_Length) or else
           (From_The_Position > To_The_Position) then
            raise Position_Error;
        else
            return The_String.The_Items(From_The_Position ..
To_The_Position);
        end if;
    end Substring_Of;

    procedure Iterate (Over_The_String : in String) is
        Continue : Boolean;
    begin
        for The_Iterator in 1 .. Over_The_String.The_Length loop
            Process(Over_The_String.The_Items(The_Iterator),
Continue);
            exit when not Continue;
        end loop;
    end Iterate;

end String_Sequential_Unbounded_Unmanaged_Iterator;
```

STRING SEQUENTIAL UNBOUNDED UNMANAGED ITERATOR

PSDL

```
TYPE String_Sequential_Unbounded_Unmanaged_Iterator
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE,
    Substring : ARRAY[ARRAY_ELEMENT : Item, ARRAY_INDEX : Positive],
    func_"<" : FUNCTION[Left : Item, Right : Item, RETURN : Boolean]
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_String : String
    OUTPUT
      The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Prepend
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_String : String,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Append
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      To_The_String : String
    OUTPUT
      To_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_String : String,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Insert
  SPECIFICATION
    INPUT
      The_Substring : Substring,
      In_The_String : String,
      At_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Delete
  SPECIFICATION
    INPUT
      In_The_String : String,
      From_The_Position : Positive,
      To_The_Position : Positive
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_String : String
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Replace
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Substring : Substring
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      In_The_String : String,
      At_The_Position : Positive,
      With_The_Item : Item
    OUTPUT
      In_The_String : String
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Substring,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : String,
      Right : Substring
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END

  OPERATOR Is_Less_Than
  SPECIFICATION
    INPUT
      Left : String,
      Right : String
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Position_Error
  END
```

```
OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Less_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : Substring,
    Right : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Greater_Than
SPECIFICATION
  INPUT
    Left : String,
    Right : Substring
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Length_Of
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Natural
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Is_Null
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Boolean
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Item_Of
SPECIFICATION
  INPUT
    The_String : String,
    At_The_Position : Positive
  OUTPUT
    Result : Item
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Substring_Of
SPECIFICATION
  INPUT
    The_String : String
  OUTPUT
    Result : Substring
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Substring_Of
SPECIFICATION
  INPUT
    The_String : String,
    From_The_Position : Positive,
    To_The_Position : Positive
  OUTPUT
    Result : Substring
  EXCEPTIONS
    Overflow, Position_Error
END

OPERATOR Iterate
SPECIFICATION
  GENERIC
    Process : PROCEDURE[The_Item : in[t : Item], Continue : out[t :
Boolean]]
  INPUT
    Over_The_String : String
  EXCEPTIONS
    Overflow, Position_Error
END

END
IMPLEMENTATION ADA String_Sequential_Unbounded_Unmanaged_Iterator
END
```

TREE ARBITRARY DOUBLE UNBOUNDED MANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
    Expected_Number_Of_Children : in Positive;
package Tree_Arbitrary_Double_Unbounded_Managed is

    type Tree is private;

    Null_Tree : constant Tree;

    procedure Copy         (From_The_Tree      : in      Tree;
                            To_The_Tree        : in out Tree);
    procedure Clear        (The_Tree           : in out Tree);
    procedure Construct    (The_Item           : in      Item;
                            And_The_Tree       : in out Tree;
                            Number_Of_Children : in      Natural;
                            On_The_Child       : in      Natural);
    procedure Set_Item     (Of_The_Tree        : in out Tree;
                            To_The_Item        : in      Item);
    procedure Swap_Child   (The_Child          : in      Positive;
                            Of_The_Tree        : in out Tree;
                            And_The_Tree       : in out Tree);

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

    procedure Is_Equal                (Left      : in Tree;
                                       Right     : in Tree;
                                       Result    : out Boolean);
    procedure Is_Null                 (The_Tree  : in Tree;
                                       Result    : out Boolean);
    procedure Item_Of                 (The_Tree  : in Tree);
                                       Result    : out Item);
    procedure Number_Of_Children_In (The_Tree  : in Tree;
                                       Result    : out Natural);
    procedure Child_Of                (The_Tree  : in Tree;
                                       The_Child : in Positive;
                                       Result    : out Tree);

--  end of modification

    function Is_Equal                 (Left      : in Tree;
                                       Right     : in Tree)      return
Boolean;
    function Is_Null                  (The_Tree  : in Tree)      return
Boolean;
    function Item_Of                  (The_Tree  : in Tree)      return
Item;
    function Number_Of_Children_In  (The_Tree  : in Tree)      return
Natural;
    function Child_Of                 (The_Tree  : in Tree;
                                       The_Child : in Positive) return
Tree;
    function Parent_Of                (The_Tree  : in Tree)      return
Tree;

    Overflow          : exception;
    Tree_Is_Null      : exception;
    Tree_Is_Not_Null  : exception;
    Not_At_Root       : exception;
    Child_Error       : exception;

private
    type Node;
    type Tree is access Node;
    Null_Tree : constant Tree := null;
end Tree_Arbitrary_Double_Unbounded_Managed;
```

TREE ARBITRARY DOUBLE UNBOUNDED MANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator,
     Storage_Manager_Sequential;
package body Tree_Arbitrary_Double_Unbounded_Managed is

    function Hash_Of (The_Child : in Positive) return Positive;

    package Children is new
      Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
        (Domain            => Positive,
         Ranges            => Tree,
         Number_Of_Buckets => Expected_Number_Of_Children,
         Hash_Of           => Hash_Of);

    type Node is
        record
            Previous     : Tree;
            The_Item     : Item;
            The_Children : Children.Map;
            Next         : Tree;
        end record;

    function Hash_Of (The_Child : in Positive) return Positive is
    begin
        return The_Child;
    end Hash_Of;

    procedure Free (The_Node : in out Node) is
    begin
        The_Node.Previous := null;
        Children.Clear(The_Node.The_Children);
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Tree) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Tree is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                (Item        => Node,
                                 Pointer     => Tree,
                                 Free        => Free,
                                 Set_Pointer => Set_Next,
                                 Pointer_Of  => Next_Of);

    procedure Copy (From_The_Tree : in     Tree;
                    To_The_Tree   : in out Tree) is
        procedure Copy_Child (The_Domain : in  Positive;
                              The_Range  : in  Tree;
                              Continue   : out Boolean) is
            Temporary_Node : Tree;
        begin
            Copy(The_Range, To_The_Tree => Temporary_Node);
            Children.Bind(The_Domain, Temporary_Node,
                          In_The_Map => To_The_Tree.The_Children);
            if Temporary_Node /= Null_Tree then
                Temporary_Node.Previous := To_The_Tree;
            end if;
            Continue := True;
        end Copy_Child;
        procedure Copy_Children is new Children.Iterate(Copy_Child);
    begin
        Clear(To_The_Tree);
        if From_The_Tree /= null then
            To_The_Tree := Node_Manager.New_Item;
            To_The_Tree.The_Item := From_The_Tree.The_Item;
            Copy_Children(From_The_Tree.The_Children);
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Tree : in out Tree) is
        procedure Clear_Child (The_Domain : in  Positive;
                               The_Range  : in  Tree;
                               Continue   : out Boolean) is
            Temporary_Node : Tree := The_Range;
        begin
            Clear(Temporary_Node);
            Continue := True;
        end Clear_Child;
        procedure Clear_Children is new Children.Iterate(Clear_Child);
    begin
        if The_Tree /= null then
            Clear_Children(The_Tree.The_Children);
            Node_Manager.Free(The_Tree);
        end if;
    end Clear;

    procedure Construct (The_Item           : in     Item;
                         And_The_Tree       : in out Tree;
                         Number_Of_Children : in     Natural;
                         On_The_Child       : in     Natural) is
        Temporary_Node : Tree;
    begin
        if Number_Of_Children = 0 then
            if And_The_Tree = null then
               And_The_Tree := Node_Manager.New_Item;
               And_The_Tree.The_Item := The_Item;
               return;
            else
                raise Tree_Is_Not_Null;
            end if;
        elsif On_The_Child > Number_Of_Children then
            raise Child_Error;
        elsif And_The_Tree = null then
            And_The_Tree := Node_Manager.New_Item;
            And_The_Tree.The_Item := The_Item;
            for Index in 1 .. Number_Of_Children loop
                Children.Bind(The_Domain    => Index,
                              And_The_Range => null,
                              In_The_Map    =>
And_The_Tree.The_Children);
            end loop;
        elsif And_The_Tree.Previous = null then
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := The_Item;
            for Index in 1 .. Number_Of_Children loop
                if Index = On_The_Child then
                    Children.Bind
                      (The_Domain    => Index,
                       And_The_Range => And_The_Tree,
                       In_The_Map    => Temporary_Node.The_Children);
                else
                    Children.Bind
                      (The_Domain    => Index,
                       And_The_Range => null,
                       In_The_Map    => Temporary_Node.The_Children);
                end if;
            end loop;
            And_The_Tree.Previous := Temporary_Node;
            And_The_Tree := Temporary_Node;
        else
            raise Not_At_Root;
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Construct;

    procedure Set_Item (Of_The_Tree : in out Tree;
                        To_The_Item : in     Item) is
    begin
        Of_The_Tree.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Set_Item;

    procedure Swap_Child (The_Child    : in     Positive;
                          Of_The_Tree  : in out Tree;
                          And_The_Tree : in out Tree) is
        Temporary_Node : Tree;
    begin
        if And_The_Tree = null then
            Temporary_Node := Children.Range_Of
                                (The_Domain => The_Child,
                                 In_The_Map =>
Of_The_Tree.The_Children);
            Children.Unbind(The_Child, Of_The_Tree.The_Children);
            Children.Bind(The_Domain    => The_Child,
                          And_The_Range => null,
                          In_The_Map    => Of_The_Tree.The_Children);
            if Temporary_Node /= null then
                Temporary_Node.Previous := null;
            end if;
            And_The_Tree := Temporary_Node;
        elsif And_The_Tree.Previous = null then
            Temporary_Node := Children.Range_Of
                                (The_Domain => The_Child,
                                 In_The_Map =>
Of_The_Tree.The_Children);
            Children.Unbind(The_Child, Of_The_Tree.The_Children);
            Children.Bind(The_Domain    => The_Child,
                          And_The_Range => And_The_Tree,
                          In_The_Map    => Of_The_Tree.The_Children);
```

```
         if Temporary_Node /= null then
             Temporary_Node.Previous := null;
         end if;
         And_The_Tree.Previous := Of_The_Tree;
         And_The_Tree := Temporary_Node;
     else
         raise Not_At_Root;
     end if;
 exception
     when Constraint_Error            =>
         raise Tree_Is_Null;
     when Children.Domain_Is_Not_Bound =>
         raise Child_Error;
 end Swap_Child;

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

 procedure Is_Equal              (Left      : in Tree;
                                  Right     : in Tree;
                                  Result    : out Boolean) is
 begin
     Result := Is_Equal(Left,Right);
 end Is_Equal;

 procedure Is_Null               (The_Tree  : in Tree;
                                  Result    : out Boolean) is
 begin
     Result := Is_Null(The_Tree);
 end Is_Null;

 procedure Item_Of               (The_Tree  : in Tree);
                                  Result    : out Item) is
 begin
     Result := Item_Of(The_Tree);
 end Item_Of;

 procedure Number_Of_Children_In (The_Tree  : in Tree;
                                  Result    : out Natural) is
 begin
     Result := Number_Of_Children_In(The_Tree);
 end Number_Of_Children_In;

 procedure Child_Of              (The_Tree  : in Tree;
                                  The_Child : in Positive;
                                  Result    : out Tree) is
 begin
     Result := Child_Of(The_Tree,The_Child);
 end Child_Of;

--  end of modification

 function Is_Equal (Left  : in Tree;
                    Right : in Tree) return Boolean is
     Trees_Are_Equal : Boolean := True;
     procedure Check_Child_Equality (The_Domain : in  Positive;
                                     The_Range  : in  Tree;
                                     Continue   : out Boolean) is
     begin
         if not Is_Equal(The_Range,
                         Children.Range_Of(The_Domain,
                                           Right.The_Children))
then
             Trees_Are_Equal := False;
             Continue := False;
         else
             Continue := True;
         end if;
     end Check_Child_Equality;
     procedure Check_Equality is new
Children.Iterate(Check_Child_Equality);
 begin
     if Left.The_Item /= Right.The_Item then
         return False;
     else
         if Children.Extent_Of(Left.The_Children) /=
            Children.Extent_Of(Right.The_Children) then
             return False;
         else
             Check_Equality(Left.The_Children);
             return Trees_Are_Equal;
         end if;
     end if;
 exception
     when Constraint_Error =>
         return (Left = Null_Tree) and (Right = Null_Tree);
 end Is_Equal;

 function Is_Null (The_Tree : in Tree) return Boolean is
 begin
     return (The_Tree = null);
 end Is_Null;

 function Item_Of (The_Tree : in Tree) return Item is
 begin
     return The_Tree.The_Item;
 exception
     when Constraint_Error =>
         raise Tree_Is_Null;
 end Item_Of;

 function Number_Of_Children_In (The_Tree : in Tree) return Natural
is
 begin
     return Children.Extent_Of(The_Tree.The_Children);
 exception
     when Constraint_Error =>
         raise Tree_Is_Null;
 end Number_Of_Children_In;

 function Child_Of (The_Tree  : in Tree;
                    The_Child : in Positive) return Tree is
 begin
     return Children.Range_Of(The_Domain => The_Child,
                              In_The_Map => The_Tree.The_Children);
 exception
     when Constraint_Error            =>
         raise Tree_Is_Null;
     when Children.Domain_Is_Not_Bound =>
         raise Child_Error;
 end Child_Of;

 function Parent_Of (The_Tree : in Tree) return Tree is
 begin
     return The_Tree.Previous;
 exception
     when Constraint_Error =>
         raise Tree_Is_Null;
 end Parent_Of;

end Tree_Arbitrary_Double_Unbounded_Managed;
```

TREE ARBITRARY DOUBLE UNBOUNDED MANAGED

PSDL

```
TYPE Tree_Arbitrary_Double_Unbounded_Managed
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Tree : Tree,
      To_The_Tree : Tree
    OUTPUT
      To_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_Tree : Tree,
      Number_Of_Children : Natural,
      On_The_Child : Natural
    OUTPUT
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      Of_The_Tree : Tree,
      To_The_Item : Item
    OUTPUT
      Of_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Swap_Child
  SPECIFICATION
    INPUT
      The_Child : Positive,
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    OUTPUT
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Tree,
      Right : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Item_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Number_Of_Children_In
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Child_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree,
      The_Child : Positive
    OUTPUT
      Result : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

END
IMPLEMENTATION ADA Tree_Arbitrary_Double_Unbounded_Managed
END
```

TREE ARBITRARY DOUBLE UNBOUNDED UNMANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
    Expected_Number_Of_Children : in Positive;
package Tree_Arbitrary_Double_Unbounded_Unmanaged is

    type Tree is private;

    Null_Tree : constant Tree;

    procedure Copy       (From_The_Tree       : in     Tree;
                          To_The_Tree         : in out Tree);
    procedure Clear      (The_Tree            : in out Tree);
    procedure Construct  (The_Item            : in     Item;
                          And_The_Tree        : in out Tree;
                          Number_Of_Children  : in     Natural;
                          On_The_Child        : in     Natural);
    procedure Set_Item   (Of_The_Tree         : in out Tree;
                          To_The_Item         : in     Item);
    procedure Swap_Child (The_Child           : in     Positive;
                          Of_The_Tree         : in out Tree;
                          And_The_Tree        : in out Tree);

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

    procedure Is_Equal              (Left      : in Tree;
                                     Right     : in Tree;
                                     Result    : out Boolean);
    procedure Is_Null               (The_Tree  : in Tree;
                                     Result    : out Boolean);
    procedure Item_Of               (The_Tree  : in Tree);
                                     Result    : out Item);
    procedure Number_Of_Children_In (The_Tree  : in Tree;
                                     Result    : out Natural);
    procedure Child_Of              (The_Tree  : in Tree;
                                     The_Child : in Positive;
                                     Result    : out Tree);

--  end of modification

    function Is_Equal              (Left      : in Tree;
                                    Right     : in Tree)      return
Boolean;
    function Is_Null               (The_Tree  : in Tree)      return
Boolean;
    function Item_Of               (The_Tree  : in Tree)      return
Item;
    function Number_Of_Children_In (The_Tree  : in Tree)      return
Natural;
    function Child_Of              (The_Tree  : in Tree;
                                    The_Child : in Positive)  return
Tree;
    function Parent_Of             (The_Tree  : in Tree)      return
Tree;

    Overflow          : exception;
    Tree_Is_Null      : exception;
    Tree_Is_Not_Null  : exception;
    Not_At_Root       : exception;
    Child_Error       : exception;

private
    type Node;
    type Tree is access Node;
    Null_Tree : constant Tree := null;
end Tree_Arbitrary_Double_Unbounded_Unmanaged;
```

TREE ARBITRARY DOUBLE UNBOUNDED UNMANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator;
package body Tree_Arbitrary_Double_Unbounded_Unmanaged is

    function Hash_Of (The_Child : in Positive) return Positive;

    package Children is new
      Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
        (Domain            => Positive,
         Ranges            => Tree,
         Number_Of_Buckets => Expected_Number_Of_Children,
         Hash_Of           => Hash_Of);

    type Node is
        record
            Previous     : Tree;
            The_Item     : Item;
            The_Children : Children.Map;
        end record;

    function Hash_Of (The_Child : in Positive) return Positive is
    begin
        return The_Child;
    end Hash_Of;

    procedure Copy (From_The_Tree : in     Tree;
                    To_The_Tree   : in out Tree) is
        procedure Copy_Child (The_Domain : in  Positive;
                              The_Range  : in  Tree;
                              Continue   : out Boolean) is
            Temporary_Node : Tree;
        begin
            Copy(The_Range, To_The_Tree => Temporary_Node);
            Children.Bind(The_Domain, Temporary_Node,
                          In_The_Map => To_The_Tree.The_Children);
            if Temporary_Node /= Null_Tree then
                Temporary_Node.Previous := To_The_Tree;
            end if;
            Continue := True;
        end Copy_Child;
        procedure Copy_Children is new Children.Iterate(Copy_Child);
    begin
        if From_The_Tree = null then
            To_The_Tree := null;
        else
            To_The_Tree := new Node;
            To_The_Tree.The_Item := From_The_Tree.The_Item;
            Copy_Children(From_The_Tree.The_Children);
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Tree : in out Tree) is
    begin
        The_Tree := null;
    end Clear;

    procedure Construct (The_Item           : in     Item;
                         And_The_Tree       : in out Tree;
                         Number_Of_Children : in     Natural;
                         On_The_Child       : in     Natural) is
        Temporary_Node : Tree;
    begin
        if Number_Of_Children = 0 then
            if And_The_Tree = null then
               And_The_Tree := new Node;
               And_The_Tree.The_Item := The_Item;
               return;
            else
                raise Tree_Is_Not_Null;
            end if;
        elsif On_The_Child > Number_Of_Children then
            raise Child_Error;
        elsif And_The_Tree = null then
            And_The_Tree := new Node;
            And_The_Tree.The_Item := The_Item;
            for Index in 1 .. Number_Of_Children loop
                Children.Bind(The_Domain    => Index,
                              And_The_Range => null,
                              In_The_Map    =>
And_The_Tree.The_Children);
            end loop;
        elsif And_The_Tree.Previous = null then
            Temporary_Node := new Node;
            Temporary_Node.The_Item := The_Item;
            for Index in 1 .. Number_Of_Children loop
                if Index = On_The_Child then
                    Children.Bind
                      (The_Domain => Index,
                       And_The_Range  => And_The_Tree,
                       In_The_Map     => Temporary_Node.The_Children);
                else
                    Children.Bind
                      (The_Domain     => Index,
                       And_The_Range => null,
                       In_The_Map     => Temporary_Node.The_Children);
                end if;
            end loop;
            And_The_Tree.Previous := Temporary_Node;
            And_The_Tree := Temporary_Node;
        else
            raise Not_At_Root;
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Construct;

    procedure Set_Item (Of_The_Tree : in out Tree;
                        To_The_Item : in     Item) is
    begin
        Of_The_Tree.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Set_Item;

    procedure Swap_Child (The_Child    : in     Positive;
                          Of_The_Tree  : in out Tree;
                          And_The_Tree : in out Tree) is
        Temporary_Node : Tree;
    begin
        if And_The_Tree = null then
            Temporary_Node := Children.Range_Of
                                (The_Domain => The_Child,
                                 In_The_Map =>
Of_The_Tree.The_Children);
            Children.Unbind(The_Child, Of_The_Tree.The_Children);
            Children.Bind(The_Domain    => The_Child,
                          And_The_Range => null,
                          In_The_Map    => Of_The_Tree.The_Children);
            if Temporary_Node /= null then
                Temporary_Node.Previous := null;
            end if;
            And_The_Tree := Temporary_Node;
        elsif And_The_Tree.Previous = null then
            Temporary_Node := Children.Range_Of
                                (The_Domain => The_Child,
                                 In_The_Map =>
Of_The_Tree.The_Children);
            Children.Unbind(The_Child, Of_The_Tree.The_Children);
            Children.Bind(The_Domain    => The_Child,
                          And_The_Range => And_The_Tree,
                          In_The_Map    => Of_The_Tree.The_Children);
            if Temporary_Node /= Null_Tree then
                Temporary_Node.Previous := null;
            end if;
            And_The_Tree.Previous := Of_The_Tree;
            And_The_Tree := Temporary_Node;
        else
            raise Not_At_Root;
        end if;
    exception
        when Constraint_Error             =>
            raise Tree_Is_Null;
        when Children.Domain_Is_Not_Bound =>
            raise Child_Error;
    end Swap_Child;

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

    procedure Is_Equal              (Left      : in Tree;
                                     Right     : in Tree;
                                     Result    : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Is_Null               (The_Tree  : in Tree;
                                     Result    : out Boolean) is
    begin
        Result := Is_Null(The_Tree);
    end Is_Null;

    procedure Item_Of               (The_Tree  : in Tree);
                                     Result    : out Item) is
    begin
        Result := Item_Of(The_Tree);
    end Item_Of;
```

```
    procedure Number_Of_Children_In (The_Tree   : in Tree;
                                     Result     : out Natural) is
    begin
        Result := Number_Of_Children_In(The_Tree);
    end Number_Of_Children_In;

    procedure Child_Of              (The_Tree  : in Tree;
                                     The_Child : in Positive;
                                     Result    : out Tree) is
    begin
        Result := Child_Of(The_Tree,The_Child);
    end Child_Of;

--  end of modification

    function Is_Equal (Left  : in Tree;
                       Right : in Tree) return Boolean is
        Trees_Are_Equal : Boolean := True;
        procedure Check_Child_Equality (The_Domain : in  Positive;
                                        The_Range  : in  Tree;
                                        Continue   : out Boolean) is
        begin
            if not Is_Equal(The_Range,
                            Children.Range_Of(The_Domain,
                                              Right.The_Children))
then
                Trees_Are_Equal := False;
                Continue := False;
            else
                Continue := True;
            end if;
        end Check_Child_Equality;
        procedure Check_Equality is new
Children.Iterate(Check_Child_Equality);
    begin
        if Left.The_Item /= Right.The_Item then
            return False;
        else
            if Children.Extent_Of(Left.The_Children) /=
               Children.Extent_Of(Right.The_Children) then
                return False;
            else
                Check_Equality(Left.The_Children);
                return Trees_Are_Equal;
            end if;
        end if;
    exception
        when Constraint_Error =>
            return (Left = Null_Tree) and (Right = Null_Tree);
    end Is_Equal;

    function Is_Null (The_Tree : in Tree) return Boolean is
    begin
        return (The_Tree = null);
    end Is_Null;

    function Item_Of (The_Tree : in Tree) return Item is
    begin
        return The_Tree.The_Item;
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Item_Of;

    function Number_Of_Children_In (The_Tree : in Tree) return Natural
is
    begin
        return Children.Extent_Of(The_Tree.The_Children);
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Number_Of_Children_In;

    function Child_Of (The_Tree  : in Tree;
                       The_Child : in Positive) return Tree is
    begin
        return Children.Range_Of(The_Domain => The_Child,
                                 In_The_Map => The_Tree.The_Children);
    exception
        when Constraint_Error            =>
            raise Tree_Is_Null;
        when Children.Domain_Is_Not_Bound =>
            raise Child_Error;
    end Child_Of;

    function Parent_Of (The_Tree : in Tree) return Tree is
    begin
        return The_Tree.Previous;
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Parent_Of;

end Tree_Arbitrary_Double_Unbounded_Unmanaged;
```

TREE ARBITRARY DOUBLE UNBOUNDED UNMANAGED

PSDL

```
TYPE Tree_Arbitrary_Double_Unbounded_Unmanaged
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Tree : Tree,
      To_The_Tree : Tree
    OUTPUT
      To_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_Tree : Tree,
      Number_Of_Children : Natural,
      On_The_Child : Natural
    OUTPUT
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      Of_The_Tree : Tree,
      To_The_Item : Item
    OUTPUT
      Of_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Swap_Child
  SPECIFICATION
    INPUT
      The_Child : Positive,
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    OUTPUT
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Tree,
      Right : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Item_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Number_Of_Children_In
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

  OPERATOR Child_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree,
      The_Child : Positive
    OUTPUT
      Result : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Not_At_Root,
Child_Error
  END

END
IMPLEMENTATION ADA Tree_Arbitrary_Double_Unbounded_Unmanaged
END
```

TREE ARBITRARY SINGLE UNBOUNDED MANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
    Expected_Number_Of_Children : in Positive;
package Tree_Arbitrary_Single_Unbounded_Managed is

    type Tree is private;

    Null_Tree : constant Tree;

    procedure Copy       (From_The_Tree      : in      Tree;
                          To_The_Tree        : in out Tree);
    procedure Clear      (The_Tree           : in out Tree);
    procedure Construct  (The_Item           : in      Item;
                          And_The_Tree       : in out Tree;
                          Number_Of_Children : in      Natural;
                          On_The_Child       : in      Natural);
    procedure Set_Item   (Of_The_Tree        : in out Tree;
                          To_The_Item        : in      Item);
    procedure Swap_Child (The_Child          : in      Positive;
                          Of_The_Tree        : in out Tree;
                          And_The_Tree       : in out Tree);

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

    procedure Is_Equal                 (Left      : in Tree;
                                        Right     : in Tree;
                                        Result    : out Boolean);
    procedure Is_Null                  (The_Tree  : in Tree;
                                        Result    : out Boolean);
    procedure Item_Of                  (The_Tree  : in Tree);
                                        Result    : out Item);
    procedure Number_Of_Children_In    (The_Tree  : in Tree;
                                        Result    : out Natural);
    procedure Child_Of                 (The_Tree  : in Tree;
                                        The_Child : in Positive;
                                        Result    : out Tree);
--  end of modification

    function Is_Equal                 (Left      : in Tree;
                                       Right     : in Tree)       return
Boolean;
    function Is_Null                  (The_Tree  : in Tree)       return
Boolean;
    function Item_Of                  (The_Tree  : in Tree)       return
Item;
    function Number_Of_Children_In    (The_Tree  : in Tree)       return
Natural;
    function Child_Of                 (The_Tree  : in Tree;
                                       The_Child : in Positive) return
Tree;

    Overflow         : exception;
    Tree_Is_Null     : exception;
    Tree_Is_Not_Null : exception;
    Child_Error      : exception;

private
    type Node;
    type Tree is access Node;
    Null_Tree : constant Tree := null;
end Tree_Arbitrary_Single_Unbounded_Managed;
```

TREE ARBITRARY SINGLE UNBOUNDED MANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator,
     Storage_Manager_Sequential;
package body Tree_Arbitrary_Single_Unbounded_Managed is

    function Hash_Of (The_Child : in Positive) return Positive;

    package Children is new
      Map_Simple_Noncached_Sequential_Unbounded_Managed_Iterator
        (Domain            => Positive,
         Ranges            => Tree,
         Number_Of_Buckets => Expected_Number_Of_Children,
         Hash_Of           => Hash_Of);

    type Node is
        record
            The_Item     : Item;
            The_Children : Children.Map;
            Next         : Tree;
        end record;

    function Hash_Of (The_Child : in Positive) return Positive is
    begin
        return The_Child;
    end Hash_Of;

    procedure Free (The_Node : in out Node) is
    begin
        Children.Clear(The_Node.The_Children);
    end Free;

    procedure Set_Next (The_Node : in out Node;
                        To_Next  : in     Tree) is
    begin
        The_Node.Next := To_Next;
    end Set_Next;

    function Next_Of (The_Node : in Node) return Tree is
    begin
        return The_Node.Next;
    end Next_Of;

    package Node_Manager is new Storage_Manager_Sequential
                                  (Item        => Node,
                                   Pointer     => Tree,
                                   Free        => Free,
                                   Set_Pointer => Set_Next,
                                   Pointer_Of  => Next_Of);

    procedure Copy (From_The_Tree : in     Tree;
                    To_The_Tree   : in out Tree) is
        procedure Copy_Child (The_Domain : in  Positive;
                              The_Range  : in  Tree;
                              Continue   : out Boolean) is
            Temporary_Node : Tree;
        begin
            Copy(The_Range, To_The_Tree => Temporary_Node);
            Children.Bind(The_Domain, Temporary_Node,
                          In_The_Map => To_The_Tree.The_Children);
            Continue := True;
        end Copy_Child;
        procedure Copy_Children is new Children.Iterate(Copy_Child);
    begin
        Clear(To_The_Tree);
        if From_The_Tree /= null then
            To_The_Tree := Node_Manager.New_Item;
            To_The_Tree.The_Item := From_The_Tree.The_Item;
            Copy_Children(From_The_Tree.The_Children);
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Tree : in out Tree) is
        procedure Clear_Child (The_Domain : in  Positive;
                               The_Range  : in  Tree;
                               Continue   : out Boolean) is
            Temporary_Node : Tree := The_Range;
        begin
            Clear(Temporary_Node);
            Continue := True;
        end Clear_Child;
        procedure Clear_Children is new Children.Iterate(Clear_Child);
    begin
        if The_Tree /= null then
            Clear_Children(The_Tree.The_Children);
            Node_Manager.Free(The_Tree);
        end if;
    end Clear;

    procedure Construct (The_Item           : in     Item;
                         And_The_Tree       : in out Tree;
                         Number_Of_Children : in     Natural;
                         On_The_Child       : in     Natural) is
        Temporary_Node : Tree;
    begin
        if Number_Of_Children = 0 then
            if And_The_Tree = null then
                And_The_Tree := Node_Manager.New_Item;
                And_The_Tree.The_Item := The_Item;
            else
                raise Tree_Is_Not_Null;
            end if;
        elsif On_The_Child > Number_Of_Children then
            raise Child_Error;
        else
            Temporary_Node := Node_Manager.New_Item;
            Temporary_Node.The_Item := The_Item;
            for Index in 1 .. Number_Of_Children loop
                if Index = On_The_Child then
                    Children.Bind
                      (The_Domain    => Index,
                       And_The_Range => And_The_Tree,
                       In_The_Map    => Temporary_Node.The_Children);
                else
                    Children.Bind
                      (The_Domain    => Index,
                       And_The_Range => null,
                       In_The_Map    => Temporary_Node.The_Children);
                end if;
            end loop;
            And_The_Tree := Temporary_Node;
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Construct;

    procedure Set_Item (Of_The_Tree : in out Tree;
                        To_The_Item : in     Item) is
    begin
        Of_The_Tree.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Set_Item;

    procedure Swap_Child (The_Child    : in     Positive;
                          Of_The_Tree  : in out Tree;
                          And_The_Tree : in out Tree) is
        Temporary_Node : Tree;
    begin
        Temporary_Node := Children.Range_Of
                            (The_Domain => The_Child,
                             In_The_Map => Of_The_Tree.The_Children);
        Children.Unbind(The_Child, Of_The_Tree.The_Children);
        Children.Bind(The_Domain    => The_Child,
                      And_The_Range => And_The_Tree,
                      In_The_Map    => Of_The_Tree.The_Children);
        And_The_Tree := Temporary_Node;
    exception
        when Constraint_Error            =>
            raise Tree_Is_Null;
        when Children.Domain_Is_Not_Bound =>
            raise Child_Error;
    end Swap_Child;

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

    procedure Is_Equal              (Left     : in Tree;
                                     Right    : in Tree;
                                     Result   : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Is_Null               (The_Tree : in Tree;
                                     Result   : out Boolean) is
    begin
        Result := Is_Null(The_Tree);
    end Is_Null;

    procedure Item_Of               (The_Tree : in Tree);
                                     Result   : out Item) is
    begin
        Result := Item_Of(The_Tree);
    end Item_Of;

    procedure Number_Of_Children_In (The_Tree : in Tree;
                                     Result   : out Natural) is
```

```
    begin
        Result := Number_Of_Children_In(The_Tree);
    end Number_Of_Children_In;

    procedure Child_Of                (The_Tree  : in Tree;
                                       The_Child : in Positive;
                                       Result    : out Tree) is
    begin
        Result := Child_Of(The_Tree,The_Child);
    end Child_Of;

--  end of modification

    function Is_Equal (Left  : in Tree;
                       Right : in Tree) return Boolean is
        Trees_Are_Equal : Boolean := True;
        procedure Check_Child_Equality (The_Domain : in  Positive;
                                        The_Range  : in  Tree;
                                        Continue   : out Boolean) is
        begin
            if not Is_Equal(The_Range,
                            Children.Range_Of(The_Domain,
                                              Right.The_Children))
then
                Trees_Are_Equal := False;
                Continue := False;
            else
                Continue := True;
            end if;
        end Check_Child_Equality;
        procedure Check_Equality is new
Children.Iterate(Check_Child_Equality);
    begin
        if Left.The_Item /= Right.The_Item then
            return False;
        else
            if Children.Extent_Of(Left.The_Children) /=
               Children.Extent_Of(Right.The_Children) then
                return False;
            else
                Check_Equality(Left.The_Children);
                return Trees_Are_Equal;
            end if;
        end if;
    exception
        when Constraint_Error =>
            return (Left = Null_Tree) and (Right = Null_Tree);
    end Is_Equal;

    function Is_Null (The_Tree : in Tree) return Boolean is
    begin
        return (The_Tree = null);
    end Is_Null;

    function Item_Of (The_Tree : in Tree) return Item is
    begin
        return The_Tree.The_Item;
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Item_Of;

    function Number_Of_Children_In (The_Tree : in Tree) return Natural
is
    begin
        return Children.Extent_Of(The_Tree.The_Children);
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Number_Of_Children_In;

    function Child_Of (The_Tree  : in Tree;
                       The_Child : in Positive) return Tree is
    begin
        return Children.Range_Of(The_Domain => The_Child,
                                 In_The_Map => The_Tree.The_Children);
    exception
        when Constraint_Error              =>
            raise Tree_Is_Null;
        when Children.Domain_Is_Not_Bound =>
            raise Child_Error;
    end Child_Of;

end Tree_Arbitrary_Single_Unbounded_Managed;
```

TREE ARBITRARY SINGLE UNBOUNDED MANAGED

PSDL

```
TYPE Tree_Arbitrary_Single_Unbounded_Managed
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Tree : Tree,
      To_The_Tree : Tree
    OUTPUT
      To_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_Tree : Tree,
      Number_Of_Children : Natural,
      On_The_Child : Natural
    OUTPUT
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      Of_The_Tree : Tree,
      To_The_Item : Item
    OUTPUT
      Of_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Swap_Child
  SPECIFICATION
    INPUT
      The_Child : Positive,
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    OUTPUT
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Tree,
      Right : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Item_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Number_Of_Children_In
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Child_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree,
      The_Child : Positive
    OUTPUT
      Result : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

END
IMPLEMENTATION ADA Tree_Arbitrary_Single_Unbounded_Managed
END
```

TREE ARBITRARY SINGLE UNBOUNDED UNMANAGED

ADA SPECIFICATIONS

```
generic
    type Item is private;
    Expected_Number_Of_Children : in Positive;
package Tree_Arbitrary_Single_Unbounded_Unmanaged is

    type Tree is private;

    Null_Tree : constant Tree;

    procedure Copy      (From_The_Tree      : in     Tree;
                         To_The_Tree        : in out Tree);
    procedure Clear     (The_Tree           : in out Tree);
    procedure Construct (The_Item           : in     Item;
                         And_The_Tree       : in out Tree;
                         Number_Of_Children : in     Natural;
                         On_The_Child       : in     Natural);
    procedure Set_Item  (Of_The_Tree        : in out Tree;
                         To_The_Item        : in     Item);
    procedure Swap_Child (The_Child         : in     Positive;
                         Of_The_Tree        : in out Tree;
                         And_The_Tree       : in out Tree);

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

    procedure Is_Equal              (Left     : in Tree;
                                     Right    : in Tree;
                                     Result   : out Boolean);
    procedure Is_Null               (The_Tree : in Tree;
                                     Result   : out Boolean);
    procedure Item_Of               (The_Tree : in Tree);
                                     Result   : out Item);
    procedure Number_Of_Children_In (The_Tree : in Tree;
                                     Result   : out Natural);
    procedure Child_Of              (The_Tree : in Tree;
                                     The_Child : in Positive;
                                     Result   : out Tree);

--  end of modification

    function Is_Equal              (Left     : in Tree;
                                    Right    : in Tree)      return
Boolean;
    function Is_Null               (The_Tree : in Tree)      return
Boolean;
    function Item_Of               (The_Tree : in Tree)      return
Item;
    function Number_Of_Children_In (The_Tree : in Tree)      return
Natural;
    function Child_Of              (The_Tree : in Tree;
                                    The_Child : in Positive) return
Tree;

    Overflow         : exception;
    Tree_Is_Null     : exception;
    Tree_Is_Not_Null : exception;
    Child_Error      : exception;

private
    type Node;
    type Tree is access Node;
    Null_Tree : constant Tree := null;
end Tree_Arbitrary_Single_Unbounded_Unmanaged;
```

TREE ARBITRARY SINGLE UNBOUNDED UNMANAGED

ADA IMPLEMENTATION

```
--
-- (C) Copyright 1986, 1987, 1988, 1989, 1990 Grady Booch
-- All Rights Reserved
--
-- Serial Number 0100219
--
--          "Restricted Rights Legend"
-- Use, duplication, or disclosure is subject to
-- restrictions as set forth in subdivision (b) (3) (ii)
-- of the rights in Technical Data and Computer
-- Software Clause of FAR 52.227-7013. Manufacturer:
-- Wizard software, 2171 S. Parfet Court, Lakewood,
-- Colorado 80227 (1-303-987-1874)
--
with Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator;
package body Tree_Arbitrary_Single_Unbounded_Unmanaged is

    function Hash_Of (The_Child : in Positive) return Positive;

    package Children is new
      Map_Simple_Noncached_Sequential_Unbounded_Unmanaged_Iterator
        (Domain            => Positive,
         Ranges            => Tree,
         Number_Of_Buckets => Expected_Number_Of_Children,
         Hash_Of           => Hash_Of);

    type Node is
        record
            The_Item     : Item;
            The_Children : Children.Map;
        end record;

    function Hash_Of (The_Child : in Positive) return Positive is
    begin
        return The_Child;
    end Hash_Of;

    procedure Copy (From_The_Tree : in     Tree;
                    To_The_Tree   : in out Tree) is
        procedure Copy_Child (The_Domain : in  Positive;
                              The_Range  : in  Tree;
                              Continue   : out Boolean) is
            Temporary_Node : Tree;
        begin
            Copy(The_Range, To_The_Tree => Temporary_Node);
            Children.Bind(The_Domain, Temporary_Node,
                          In_The_Map => To_The_Tree.The_Children);
            Continue := True;
        end Copy_Child;
        procedure Copy_Children is new Children.Iterate(Copy_Child);
    begin
        if From_The_Tree = null then
            To_The_Tree := null;
        else
            To_The_Tree := new Node;
            To_The_Tree.The_Item := From_The_Tree.The_Item;
            Copy_Children(From_The_Tree.The_Children);
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Copy;

    procedure Clear (The_Tree : in out Tree) is
    begin
        The_Tree := null;
    end Clear;

    procedure Construct (The_Item           : in     Item;
                         And_The_Tree       : in out Tree;
                         Number_Of_Children : in     Natural;
                         On_The_Child       : in     Natural) is
        Temporary_Node : Tree;
    begin
        if Number_Of_Children = 0 then
            if And_The_Tree = null then
               And_The_Tree := new Node;
               And_The_Tree.The_Item := The_Item;
               return;
            else
               raise Tree_Is_Not_Null;
            end if;
        elsif On_The_Child > Number_Of_Children then
            raise Child_Error;
        else
            Temporary_Node := new Node;
            Temporary_Node.The_Item := The_Item;
            for Index in 1 .. Number_Of_Children loop
                if Index = On_The_Child then
                    Children.Bind
                      (The_Domain    => Index,
                       And_The_Range => And_The_Tree,
                       In_The_Map    => Temporary_Node.The_Children);
                else
                    Children.Bind
                      (The_Domain    => Index,
                       And_The_Range => null,
                       In_The_Map    => Temporary_Node.The_Children);
                end if;
            end loop;
            And_The_Tree := Temporary_Node;
        end if;
    exception
        when Storage_Error | Children.Overflow =>
            raise Overflow;
    end Construct;

    procedure Set_Item (Of_The_Tree : in out Tree;
                        To_The_Item : in     Item) is
    begin
        Of_The_Tree.The_Item := To_The_Item;
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Set_Item;

    procedure Swap_Child (The_Child    : in     Positive;
                          Of_The_Tree  : in out Tree;
                          And_The_Tree : in out Tree) is
        Temporary_Node : Tree;
    begin
        Temporary_Node := Children.Range_Of
                            (The_Domain => The_Child,
                             In_The_Map => Of_The_Tree.The_Children);
        Children.Unbind(The_Child, Of_The_Tree.The_Children);
        Children.Bind(The_Domain    => The_Child,
                      And_The_Range => And_The_Tree,
                      In_The_Map    => Of_The_Tree.The_Children);
        And_The_Tree := Temporary_Node;
    exception
        when Constraint_Error             =>
            raise Tree_Is_Null;
        when Children.Domain_Is_Not_Bound =>
            raise Child_Error;
    end Swap_Child;

--  modified by Tuan Nguyen
--  25 December 1995
--  adding procedures to replace functions

    procedure Is_Equal              (Left      : in Tree;
                                     Right     : in Tree;
                                     Result    : out Boolean) is
    begin
        Result := Is_Equal(Left,Right);
    end Is_Equal;

    procedure Is_Null               (The_Tree  : in Tree;
                                     Result    : out Boolean) is
    begin
        Result := Is_Null(The_Tree);
    end Is_Null;

    procedure Item_Of               (The_Tree  : in Tree);
                                     Result    : out Item) is
    begin
        Result := Item_Of(The_Tree);
    end Item_Of;

    procedure Number_Of_Children_In (The_Tree  : in Tree;
                                     Result    : out Natural) is
    begin
        Result := Number_Of_Children_In(The_Tree);
    end Number_Of_Children_In;

    procedure Child_Of              (The_Tree  : in Tree;
                                     The_Child : in Positive;
                                     Result    : out Tree) is
    begin
        Result := Child_Of(The_Tree,The_Child);
    end Child_Of;

--  end of modification

    function Is_Equal (Left  : in Tree;
                       Right : in Tree) return Boolean is
        Trees_Are_Equal : Boolean := True;
        procedure Check_Child_Equality (The_Domain : in  Positive;
                                        The_Range  : in  Tree;
                                        Continue   : out Boolean) is
        begin
            if not Is_Equal(The_Range,
                            Children.Range_Of(The_Domain,
                                              Right.The_Children))
then
                Trees_Are_Equal := False;
                Continue := False;
            else
                Continue := True;
            end if;
        end Check_Child_Equality;
        procedure Check_Equality is new
Children.Iterate(Check_Child_Equality);
    begin
        if Left.The_Item /= Right.The_Item then
            return False;
```

```
    else
        if Children.Extent_Of(Left.The_Children) /=
           Children.Extent_Of(Right.The_Children) then
            return False;
        else
            Check_Equality(Left.The_Children);
            return Trees_Are_Equal;
        end if;
    end if;
exception
    when Constraint_Error =>
        return (Left = Null_Tree) and (Right = Null_Tree);
end Is_Equal;

function Is_Null (The_Tree : in Tree) return Boolean is
begin
    return (The_Tree = null);
end Is_Null;

function Item_Of (The_Tree : in Tree) return Item is
begin
    return The_Tree.The_Item;
exception
    when Constraint_Error =>
        raise Tree_Is_Null;
```

```
    end Item_Of;

    function Number_Of_Children_In (The_Tree : in Tree) return Natural
is
    begin
        return Children.Extent_Of(The_Tree.The_Children);
    exception
        when Constraint_Error =>
            raise Tree_Is_Null;
    end Number_Of_Children_In;

    function Child_Of (The_Tree  : in Tree;
                       The_Child : in Positive) return Tree is
    begin
        return Children.Range_Of(The_Domain => The_Child,
                                 In_The_Map => The_Tree.The_Children);
    exception
        when Constraint_Error              =>
            raise Tree_Is_Null;
        when Children.Domain_Is_Not_Bound =>
            raise Child_Error;
    end Child_Of;

end Tree_Arbitrary_Single_Unbounded_Unmanaged;
```

TREE ARBITRARY SINGLE UNBOUNDED UNMANAGED

PSDL

```
TYPE Tree_Arbitrary_Single_Unbounded_Unmanaged
SPECIFICATION
  GENERIC
    Item : PRIVATE_TYPE
  OPERATOR Copy
  SPECIFICATION
    INPUT
      From_The_Tree : Tree,
      To_The_Tree : Tree
    OUTPUT
      To_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Clear
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Construct
  SPECIFICATION
    INPUT
      The_Item : Item,
      And_The_Tree : Tree,
      Number_Of_Children : Natural,
      On_The_Child : Natural
    OUTPUT
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Set_Item
  SPECIFICATION
    INPUT
      Of_The_Tree : Tree,
      To_The_Item : Item
    OUTPUT
      Of_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Swap_Child
  SPECIFICATION
    INPUT
      The_Child : Positive,
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    OUTPUT
      Of_The_Tree : Tree,
      And_The_Tree : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Is_Equal
  SPECIFICATION
    INPUT
      Left : Tree,
      Right : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Is_Null
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Boolean
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Item_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Item
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Number_Of_Children_In
  SPECIFICATION
    INPUT
      The_Tree : Tree
    OUTPUT
      Result : Natural
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

  OPERATOR Child_Of
  SPECIFICATION
    INPUT
      The_Tree : Tree,
      The_Child : Positive
    OUTPUT
      Result : Tree
    EXCEPTIONS
      Overflow, Tree_Is_Null, Tree_Is_Not_Null, Child_Error
  END

END
IMPLEMENTATION ADA Tree_Arbitrary_Single_Unbounded_Unmanaged
END
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center 8725 John J. Kingman Road, Suite 0944 Fort Belvoir, VA 22060-6218	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Road Monterey, CA 93943	2
3.	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
4.	Dr. Ted Lewis, Chairman, Code CS/L Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
5.	Chief of Naval Research 800 N. Quincy Street Arlington, VA 22217	1
6.	Dr. Luqi, Code CS/Lq Naval Postgraduate School Computer Science Department Monterey, CA 93943-5100	20
7.	LT Tuan A. Nguyen 3673 Henrico Street Norfolk, VA 23513	3
8.	Ada Joint Program Office OUSDRE (R&AT) The Pentagon Washington, DC 20301	1
9.	Atlantic Intelligence Command 7941 Blandy Road Suite 100 Norfolk, VA 23551	1